SM4及其软件优化

> 作者：
>
> 宋诺金 202000460074

总述

本次实验主要有以下目的：

1. 基础位运算SM4思考与实现
    a. 字符串与位运算对比；
    b. 非宏定义版本keygen；
    c. 非宏定义版本crypt；
2. 论证并实现CPU上SM4优化
    a. 宏定义免过程调用
    b. 多线程
    c. SIMD
    d. 查表优化
    e. 循环展开
    f. 流水线：bitslice
3. 综合优化
    a. latency分析
    b. throughput分析
    c. 与openssl的对比

基础实现（验证正确性）

密钥生成

原初代码（省略CK与S盒）

```
#include<iostream>
#include<string.h>
#include<string>
#include<sstream>
#include<bitset>
using namespace std;
```

```
 string hextobit(long long x)
{
        bitset<32> bit(x);
        string b = bit.to_string();
        return b;}
 int CK_B[32][32] = { 0 };
 void bornCKB(const long long CK[32])
{
        for (int i = 0; i < 32; i++)
        {
                string s = hextobit(CK[i]);
            const char* p = s.data();
                for (int j = 0; j < 32; j++)
                {
                        CK_B[i][j] = p[j]-48;
                }
        }
}
 void strtobit(int* out, char* in) {
        int i, j, k;
        for (i = 0; i <128; i++)
        {
                out[i] = 0;
        }
        for (i = 0; i < 32; i++)
        {
                if (in[i] == 'a') j = 10;
                else if (in[i] == 'b') j = 11;
                else if (in[i] == 'c') j = 12;
                else if (in[i] == 'd') j = 13;
                else if (in[i] == 'e') j = 14;
                else if (in[i] == 'f') j = 15;
                else j = in[i] - 48;
                k = 4 * i + 3;
                while (k + 1 - 4 * i)
                {
                        out[k] = j % 2;
                        j = j / 2;
                        k--;
                }
        }
}
 void XOR(int* out, int* in1, int* in2, int len)
{
        for (int i = 0; i < len; i++)
        {
```

```
54                  out[i] = in1[i] ^ in2[i];
55          }
56  }
57   void leftmov(int* array, int n)  {
58          int temp[100] = { 0 };
59          for (int i = 0; i < n; i++)
60          {
61                  temp[i] = array[i];
62          }
63          for (int i = 0; i < 32 - n; i++)
64          {
65                  array[i] = array[i + n];
66          }
67          for (int i = 0; i < n; i++)
68          {
69                  array[32 - n + i] = temp[i];
70          }
71  }
72   void T(int out[32], int in[32],const int S_box[16][16])
73  {
74          /******************非线性S盒**********************/
75          int s[4][2] = { 0 };
76          int S[4] = { 0 };
77          for (int i = 0; i < 4; i++)
78          {
79                  int sum1 = in[i * 8] * 8 + in[i * 8 + 1] * 4 + in[i * 8 + 2] * 2
80                  s[i][0] = sum1;
81                  int sum2 = in[i * 8 + 4] * 8 + in[i * 8 + 5] * 4 + in[i * 8 + 6]
82                  s[i][1] = sum2;
83          }
84          for (int i = 0; i < 4; i++)
85          {
86                  S[i] = S_box[s[i][0]][s[i][1]];
87          }
88          int SinT[32] = { 0 };
89          for (int i = 0; i < 4; i++)
90          {
91                  for (int j = 1; j <= 8; j++)
92                  {
93                          if (S[i] != 0)
94                          {
95                                  SinT[(i + 1) * 8 - j] = S[i] % 2;
96                                  S[i] = S[i] / 2;
97                          }
98                          else if (S[i] == 0)
99                          {
100                                 SinT[(i + 1) * 8 - j] = 0;
```

```
                        }
                }
        }
        /*****************线性移位异或操作**********************/
        int B[32] = { 0 };
        int B13[32] = { 0 };
        int B23[32] = { 0 };
        for (int i = 0; i < 32; i++)
        {
                B[i] = SinT[i];
        }
        leftmov(SinT, 13);
        for (int i = 0; i < 32; i++)
        {
                B13[i] = SinT[i];
        }
        leftmov(SinT, 10);
        for (int i = 0; i < 32; i++)
        {
                B23[i] = SinT[i];
        }
        int unout[32] = { 0 };
        XOR(unout, B, B13, 32);
        XOR(out, unout, B23, 32);}
 void bornkey(int outKEY[32][32],char key[33])
{
        int keyb[128] = { 0 };
        strtobit(keyb, key);
        int FK[128] = { 1,0,1,0,0,0,1,1,1,0,1,1,0,0,0,1,1,0,1,1,1,0,1,0,1,1,0,0,
                        0,1,0,1,0,1,1,0,1,0,1,0,1,0,1,0,0,0,1,1,0,0,1,1,0,1,0,1,
                        0,1,1,0,0,1,1,1,0,1,1,1,1,1,0,1,1,0,0,1,0,0,0,1,1,0,0,1,
                        1,0,1,1,0,0,1,0,0,1,1,1,0,0,0,0,0,0,1,0,0,0,1,0,1,1,0,1,
        int keybx[128] = { 0 };
        XOR(keybx, keyb, FK, 128);
        int KEY[36][32] = { 0 };
        for (int i = 0; i < 4; i++)
        {
                for (int j = 0; j < 32; j++)
                {
                        KEY[i][j] = keybx[32 * i + j];
                }
        }
        int a = 0;
        for (int i = 0; i < 32; i++)
        {
                int un1[32] = { 0 };
                int un2[32] = { 0 };
```

```
148                 int un3[32] = { 0 };
149                 XOR(un1, KEY[a + 1], KEY[a + 2], 32);
150                 XOR(un2, un1, KEY[a + 3], 32);
151                 XOR(un3, un2, CK_B[a], 32);
152                 int un4[32] = { 0 };
153                 T(un4, un3, S_box);
154                 XOR(KEY[a + 4], un4, KEY[a],32);
155                 a++;
156         }
157         for (int i = 0; i < 32; i++)
158         {
159                 for (int j = 0; j < 32; j++)
160                 {
161                         outKEY[i][j] = KEY[i + 4][j];
162                 }
163         }
164 }
165  int main()
166 {
167         char key[33] = { 0 };
168         gets_s(key);
169         bornCKB(CK);
170         int KEY[32][32] = { 0 };
171         bornkey(KEY, key);
172         for (int i = 0; i < 32; i++)
173         {
174                 for (int j = 0; j < 32; j++)
175                 {
176                         cout << KEY[i][j];
177                 }
178                 cout << endl;
179         }
180         }
```

由gets_s()输入十六进制密钥，最后将32个四字密钥存入int[32][32]类型的数组中作为轮密钥

然后针对代码算法细节，我们进行逐步优化：

1. 我们决定考虑到const对数据的影响以及接口问题，我们将char与int的改用uint8_t与uint32_t。
2. 考虑到原代码涉及大量转换二进制数组循环异或的问题，我们采取直接十六进制进行按位异或的办法进行省略优化。
3. 在S盒查找细节等处，我们从原来按位乘法加和，转换为将int类型的S盒行数等于原16进制数比16，将int类型的S盒列数等于原16进制数模16，同时并行计算以获得更快的查找速度。

```
1  uint32_t S(uint8_t a[])
```

```
{
        uint8_t* p = new uint8_t[4];//8AD24122//并行查表
        p[0] = S_box[cal_x(a[0])][cal_y(a[0])];
        p[1] = S_box[cal_x(a[1])][cal_y(a[1])];
        p[2] = S_box[cal_x(a[2])][cal_y(a[2])];
        p[3] = S_box[cal_x(a[3])][cal_y(a[3])];
        uint32_t sum = p[3] + (p[2] << 8) + (p[1] << 16) + (p[0] << 24);
        return sum;}
```

4. T变换里的线性移位（B函数）操作我们直接采用C++的<<和>>移位操作，并将整个过程直接编写成函数输出结果。

```
uint32_t L(uint32_t B)
{
        uint32_t B2 = (B << 23) ^ (B >> 32 - 23);
        uint32_t B1 = (B << 13) ^ (B >> 32 - 13);
        uint32_t B0 = B ^ B1 ^ B2;
        return B0;
}
```

5. 最后汇总到密钥生成总函数里，并选择在main函数里循环生成密钥减少函数调用次数。

修改后代码（省略CK和S盒）：

```
#include<iostream>
#include<stdint.h>
#include<cmath>
using namespace std;
uint32_t FK[4] = { 0xa3b1bac6,0x56aa3350,0x677d9197,0xb27022dc };
uint32_t roundkey[36] = { 0 };
uint32_t key[4] = { 0X01234567,0X89ABCDEF,0XFEDCBA98,0X76543210 };
uint32_t* k03 = new uint32_t[4];void gen_k0(uint32_t key[])
{
        roundkey[0] = FK[0] ^ key[0];
        roundkey[1] = FK[1] ^ key[1];
        roundkey[2] = FK[2] ^ key[2];
        roundkey[3] = FK[3] ^ key[3];}
int cal_x(uint8_t a)
{
        int b = a / 16;
        return b;}
int cal_y(uint8_t a)
{
```

```
        int b = a % 16;
        return b;}
uint32_t S(uint8_t a[])
{
        uint8_t* p = new uint8_t[4];//8AD24122//并行查表
        p[0] = S_box[cal_x(a[0])][cal_y(a[0])];
        p[1] = S_box[cal_x(a[1])][cal_y(a[1])];
        p[2] = S_box[cal_x(a[2])][cal_y(a[2])];
        p[3] = S_box[cal_x(a[3])][cal_y(a[3])];
        uint32_t sum = p[3] + (p[2] << 8) + (p[1] << 16) + (p[0] << 24);
        return sum;}
uint32_t L(uint32_t B)
{
        uint32_t B2 = (B << 23) ^ (B >> 32 - 23);
        uint32_t B1 = (B << 13) ^ (B >> 32 - 13);
        uint32_t B0 = B ^ B1 ^ B2;
        return B0;}
uint32_t gen_key(uint32_t k0, uint32_t k1, uint32_t k2, uint32_t k3, uint32_t ck)
{
        uint32_t mid = k1 ^ k2 ^ k3 ^ ck;
        uint8_t* mid1 = new uint8_t[4];
        mid1[0] = mid >> 24;
        mid1[1] = mid >> 16;
        mid1[2] = mid >> 8;
        mid1[3] = mid;
        uint32_t SR = S(mid1);
        uint32_t ST = L(SR);
        uint32_t RE = ST ^ k0;
        return RE;
}
 int main()
{
        gen_k0(key);
        for (int i = 0; i < 32; i++)
        {
                roundkey[i + 4] = gen_key(roundkey[i], roundkey[i + 1], roundkey[
                printf("%02X", roundkey[i + 4]);
                cout << endl;
        }
}
```

粗略测量发现，通过修改算法代码得到的优化倍数接近四倍。

加密算法

原初版本（省略S盒）

```
void uchr2uint(uint8_t* in, uint32_t* out)
{
        int i = 0;
        *out = 0;
        for (i = 0; i < 4; i++)
                *out = ((uint32_t)in[i] << (24 - i * 8)) ^ *out;
}

void uint2uchr(uint32_t in, uint8_t* out)
{
        int i = 0;
        for (i = 0; i < 4; i++)
                *(out + i) = (uint8_t)(in >> (24 - i * 8));
}

uint32_t left_move(uint32_t data, int length)
{
        uint32_t result = 0;
        result = (data << length) ^ (data >> (32 - length));
        return result;
}
void round_fun(uint32_t* x, uint32_t k) {
        uint32_t A = x[1] ^ x[2] ^ x[3] ^ k;
        uint8_t A_chr[4];
        uint2uchr(A, A_chr);
        for (int i = 0; i < 4; i++) {
                uint8_t row = A_chr[i] >> 4;
                uint8_t col = A_chr[i] & 0b1111;
                A_chr[i] = S_Box[16 * row + col];
        }
        uint32_t B;
        uchr2uint(A_chr, &B);
        uint32_t temp = x[0] ^ B ^ left_move(B, 2) ^ left_move(B, 10) ^ left_move
        x[0] = x[1];
        x[1] = x[2];
        x[2] = x[3];
        x[3] = temp;
}
```

原初版本中分别实现了将uint32_t与uint8_t数组进行互相转换的函数（用于S盒的输入输出拆分）、循环左移的函数（用于L函数操作）以及一个完整的轮函数。考虑到函数的调用会消耗相当一部分时间，后期改进时将上述函数整合为一个内部没有函数调用的轮函数。

修改后代码（省略S盒）

```
void round_fun(uint32_t* x, uint32_t k) {
        uint32_t A = x[1] ^ x[2] ^ x[3] ^ k;
    uint8_t A_chr0, A_chr1, A_chr2, A_chr3;
    A_chr0 = A >> 24;
    A_chr1 = A >> 16;
    A_chr2 = A >> 8;
    A_chr3 = A;
    A_chr0 = S_box[(A_chr0 >> 4)][A_chr0 & 0b1111];
    A_chr1 = S_box[(A_chr1 >> 4)][A_chr1 & 0b1111];
    A_chr2 = S_box[(A_chr2 >> 4)][A_chr2 & 0b1111];
    A_chr3 = S_box[(A_chr3 >> 4)][A_chr3 & 0b1111];
    uint32_t B = (A_chr0 << 24) ^ (A_chr1 << 16) ^ (A_chr2 << 8) ^ A_chr3;
    uint32_t B2 = (B << 2) ^ (B >> 30);
    uint32_t B10 = (B << 10) ^ (B >> 22);
    uint32_t B18 = (B << 18) ^ (B >> 14);
    uint32_t B24 = (B << 24) ^ (B >> 8);
    uint32_t temp = x[0] ^ B ^ B2 ^ B10 ^ B18 ^ B24;
        x[0] = x[1];
        x[1] = x[2];
        x[2] = x[3];
        x[3] = temp;
```

结果

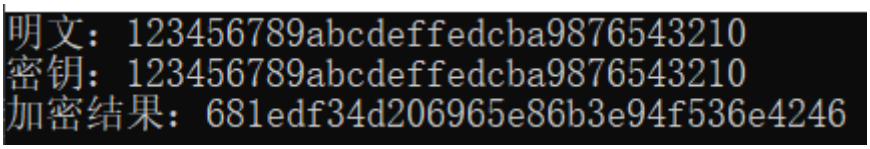
正确性展示

将密钥生成函数与轮函数整合：

```
/*生成所有轮密钥*/
void get_rnd_key(uint32_t* key,uint32_t* roundkey) {
    roundkey[0] = FK[0] ^ key[0];
    roundkey[1] = FK[1] ^ key[1];
    roundkey[2] = FK[2] ^ key[2];
    roundkey[3] = FK[3] ^ key[3];
    for (int i = 4; i < 36; i++) {
        roundkey[i] = gen_key(roundkey[i - 4], roundkey[i - 3], roundkey[i - 2],
        }
    }
int main() {
    uint32_t input[4] = { 0x01234567,0x89abcdef,0xfedcba98,0x76543210 };
    uint32_t output[4] = { 0x0,0x0,0x0,0x0 };
    uint32_t key[4] = { 0x01234567,0x89abcdef,0xfedcba98,0x76543210 };
    uint32_t roundkey[36];
    printf("明文: %02x%02x%02x%02x\n", input[0], input[1], input[2], input[3]);
    printf("密钥: %02x%02x%02x%02x\n", key[0], key[1], key[2], key[3]);
```

```
    get_rnd_key(key, roundkey);
    for (int i_0 = 0; i_0 < 32; i_0++) {
        round_fun(input, roundkey[i_0 + 4]);
    };
    output[0] = input[3];
    output[1] = input[2];
    output[2] = input[1];
    output[3] = input[0];
    printf("加密结果: %02x%02x%02x%02x\n", output[0], output[1], output[2], output
    return 0;
}
```

运行结果如下：



与标准数据对比：

4. 测试数据

数据来自《密码学引论》第三版武汉大学出版社中关于SM4算法的描述。

```
明文:   01234567 89abcdef fedcba98 76543210
密钥:   01234567 89abcdef fedcba98 76543210
密文:   681edf34 d206965e 86b3e94f 536e4246
```

这里没有使用openssl是因为openssl目前的版本强制要求使用 `-pbkdf2` 函数对输入的key进行处理，否则会出问题，所以我们找了标准数据进行比对。

基础效率分析

对于第一版，确定正确性后，我们测量运行时间，得到万次生成的平均时间为0.395s，吞吐量约为3.09Mbps。

对于第二版，确定正确性后，我们测量运行时间，得到万次生成的平均时间为0.105s，吞吐量约为11.63Mbps。

程序优化

基本优化

基本思路

在前文中我们实际封装了很多函数，如L，T这些函数。但增加过程调用是会增加额外开销的，而基础部件的调用次数又很多，会增加很多无谓的开销。因而我们可以使用宏定义，尽可能减少额外开销。

实验程序

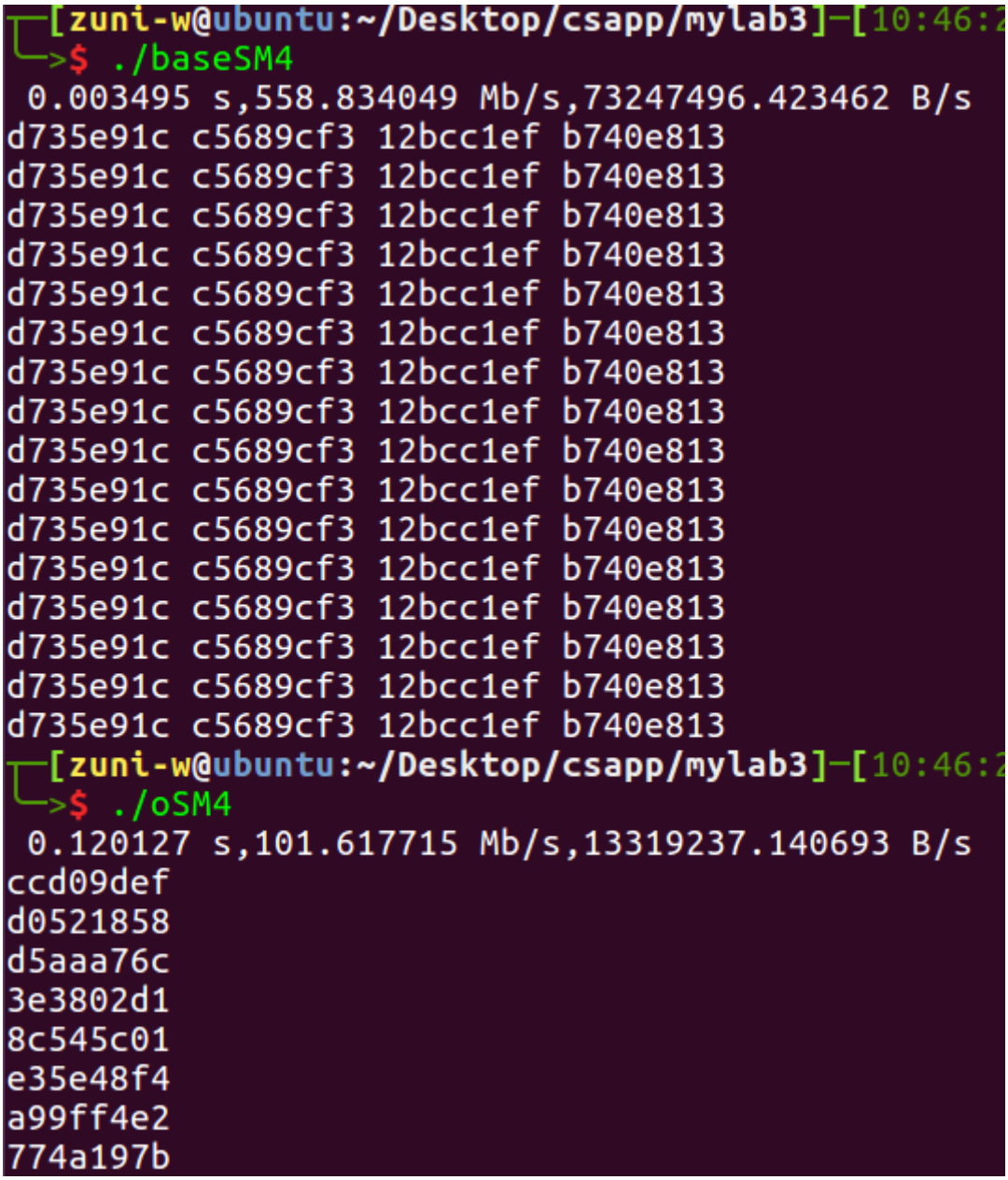
```
#define rot(x,t,n) (((x)<<(n))|((x)>>((t)-(n))))
#define u32 unsigned int
#define u8 unsigned char

void enc(u32 m[],u32* rk)
{

    for(int i=0;i<32;i++)
    {
        register u32 tmp=m[(i+1)&3]^m[(i+2)&3]^m[(i+3)&3]^*rk;
        tmp=(Sbox[(tmp>>24)&0xff]<<24)|
            (Sbox[(tmp>>16)&0xff]<<16)|
            (Sbox[(tmp>>8)&0xff]<<8)|
            (Sbox[tmp&0xff]);
        rk++;
        m[i&3]=tmp^rot(tmp,32,2)^rot(tmp,32,10)^rot(tmp,32,18)^rot(tmp,32,24)^(m[
    }
    *(__m128i*)m= _mm_shuffle_epi32(*(__m128i*)m,(0<<6)|(1<<4)|(2<<2)|3);
}
```

说明：rk是迭代完成的轮密钥，shuffle是进行重排，实际对效率影响不大。

测试程序

```
    u32* rk=keygen(key[0]);
    u32 p=1000;
    t1=clock();
    for(int i=0;i<p;i++)
    {
        for(int j=0;j<16;j+=1)
        enc(key[j],rk);
    }
    t2=clock();
    printf(" %f s,%f Mb/s,%f B/s\n",(double)(t2-t1)/CLOCKS_PER_SEC,(128.0*p*16)/(
```

实验效果



在相同环境下，可以发现优化了5倍左右。吞吐量随数据量变化的具体数据如下表，可以发现数据比较稳定。

	100	500	1000	5000	10000	50000	100000
oSM4	96	106	92	98	99	102	101
base	567	560	553	549	550	534	544

多线程优化

本部分由金周泉完成

函数内部优化

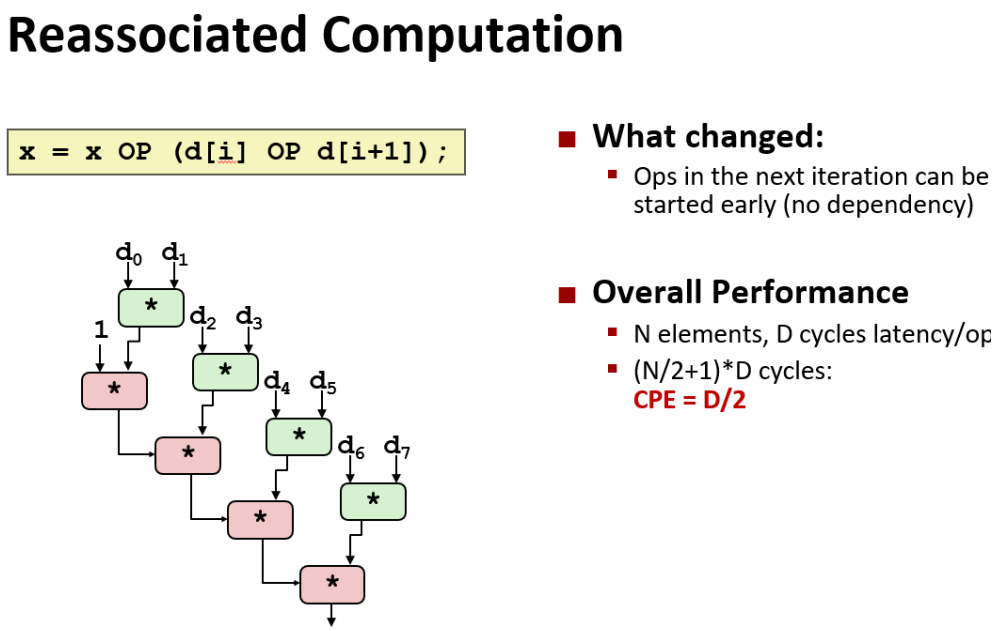
在程序完成伊始，我们本打算于函数内部进行多线程优化，但是出现以下的问题：

以密钥生成为例。我们最初的想法是利用多线程同时生成密钥。但是我们在过程中发现，对于每一个密钥k而言，我们需要K-4,K-3,K-2,K-1的密钥才能将第k个密钥生成出来。所以此时我们并不能利用多线程将密钥同步进行生成。

```
uint32_t gen_key(uint32_t k0, uint32_t k1, uint32_t k2, uint32_t k3, uint32_t ck)
{
    uint32_t mid = k1 ^ k2 ^ k3 ^ ck;
    uint8_t* mid1 = new uint8_t[4];
    mid1[0] = mid >> 24;
    mid1[1] = mid >> 16;
    mid1[2] = mid >> 8;
    mid1[3] = mid;
    uint32_t SR = S(mid1);
    uint32_t ST = L(SR);
    uint32_t RE = ST ^ k0;
    return RE;
    //mid1[0] = mid % 256;
}
```

（密钥生成过程，可以看出，每一个轮密钥都需要之前的密钥进行生成）

那么，我们是否能利用课上所讲的乘法的例子，利用两个线程，在前一个密钥生成的同时，对后面的密钥生成做一个“预处理”呢？



（课上所示例子）

答案依旧是不能。因为对于每一个轮密钥生成而言，首先第一步要做的就是要对于前四个密钥和CK进行一个异或操作，所以我们无法在不使用前一个密钥之前，对于后面的密钥去进行一个“预处理”操作。所以对于函数内部使用多线程的方式，我们不做更多研究，进而将研究重点放在对于多数据利用多线程进行加速操作。

```
uint32_t gen_key(uint32_t k0, uint32_t k1, uint32_t k2, uint32_t k3, uint32_t ck)
{
    uint32_t mid = k1 ^ k2 ^ k3 ^ ck;
```

（如图所示，在轮密钥生成的第一步，我们需要将所使用的密钥全部进行异或操作）

函数外部优化

对于函数外部优化，我们对于多数据优化进行测速

多数据优化对比

基本思想

对于多组数据使用多线程而言，我们需要将输入数据按照二维数组进行给出，并让每一个线程按照行标寻找自己所需要加密对应的数据即可。而此时由于我们需要用多线程同时对input数组进行访问，所所以我们需要将input与output从栈中拿出，放到堆中进行访问。

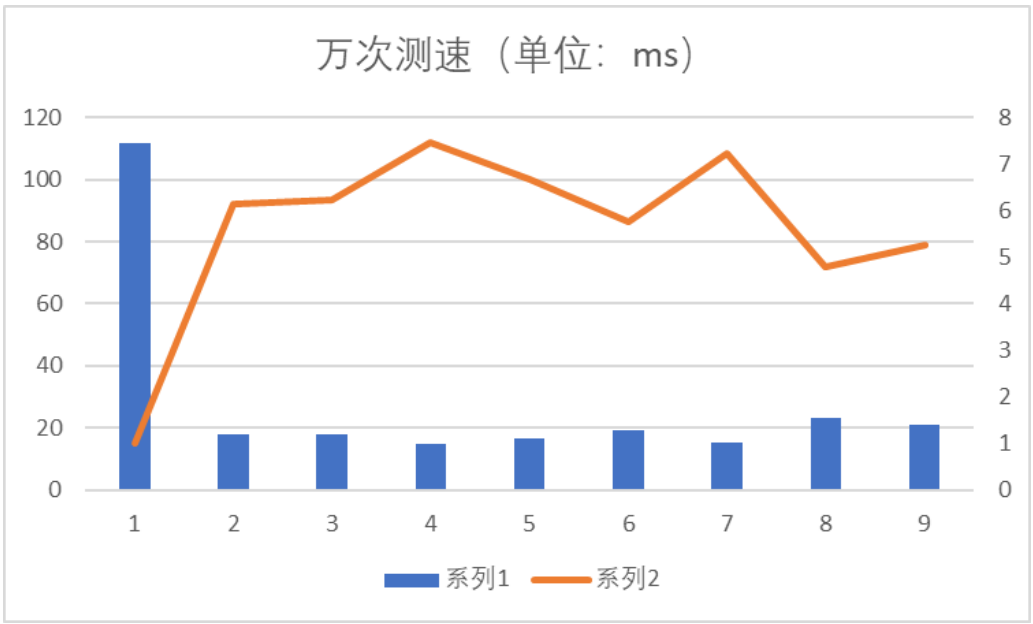
改动代码实现

```
void multi_thread(int &num)
{
        for (num; num < 10000; num += 4)
        {
                for (int i_0 = 0; i_0 < 32; i_0++)
                {
                        round_fun(input, roundkey[i_0 + 4]);
                }
        }
}
```

（此处我们就将多线程通过函数调用实现）

性能分析

我们截取了单次加密、2线程、4线程、8线程、12线程以及16线程、20线程、24线程、3线程分别进行测试，测试结果如图所：



（从左到右依次为1.2.4.8.12.16.20.24.30线程加速与加速比）

可以看出，在8线程时其加速效果最好，但多线程普遍而言加速效果并非十分理想。猜测原因如下：

1.加密速度较快，线程创建速度不可被忽略

2.加密时对于input同时访问，线程之间对于共同变量的访问导致无法并发完成。

总结：

对于本次实验而言，多线程虽然能够有一定的优化，但是总体而言优化效果并非特别理想；其加密速度较快导致线程创建时间不可被忽略与访问相同的数据都对多线程的优化产生了较大的影响。

SIMD优化

（部分参考中国科学院大学学报）

简介

SIMD(**Single Instruction Multiple Data**)即单指令流多数据流，是一种采用一个控制器来控制多个处理器，同时对一组数据（又称“数据向量”）中的每一个分别执行相同的操作从而实现空间上的并行性的技术。简单来说就是一个指令能够同时处理多个数据。

下表中展示了本文实现SM4使用的AVX2指令和对应的C/C++窗口：

AVX2指令	C/C++接口	功能描述
vpand	_mm256_and_si256	256-bit逻辑与
vpxor	_mm256_xor_si256	256-bit异或
vpsrld	_mm256_srli_epi32	8道32-bit右移
vpslld	_mm256_slli_epi32	8道32-bit左移
vpshufb	_mm256_shuffle_epi8	字节置换
vpgatherdd	_mm_i32gather_epi32	4道32-bit查表
vpgatherdd	_mm256_i32gather_epi32	8道32-bit查表

向量查表指令：vpgatherdd指令是AVX2扩展AVX最重要的指令之一，该指令可实现4/8道32-bit字并行查表。

基本思路

SM4消息存储格式

将n组128-bit SM4明文消息记为Pi ($0 \leqslant i < n$, n=4, 8)。需将Pi装载到4个SIMD寄存器R0、R1、R2、R3中。装载规则如下

$$R_k[i] \leftarrow P_i[k], 0 \leq i < n, 0 \leq k < 4,$$

其中，*Rk*[*i*]表示*Rk*寄存器中第*i*个32-bit字位置，*Pi*[*k*]表示明文消息*Pi*中第*k*个32-bit字的内容。即*Rk*寄存器依次存储着所有*n*组明文消息的第*k*个32-bit字内容。

AVX2指令支持4/8道32-bit字向量查表操作，因此装载*n*组SM4明文消息可通过向量查表指令vpgatherdd完成。

实现代码如下：

```
__m128i id, R0, R1, R2, R3;
vindex=_mm_setr_epi32(0, 4, 8, 12);
R0=_mm_i32gather_epi32((int*)(p + 4 * 0), vindex, 4);
R1=_mm_i32gather_epi32((int*)(p + 4 * 1), vindex, 4);
R2=_mm_i32gather_epi32((int*)(p + 4 * 2), vindex, 4);
R3=_mm_i32gather_epi32((int*)(p + 4 * 3), vindex, 4);
```

装载轮密钥

SM4轮函数中包含轮密钥层变换，因此为实现*n*组消息并行加/解密须将*n*个32-bit轮密钥装载到SIMD寄存器中。将32个轮密钥装载到SIMD寄存器有2种方式：1)每次轮变换时使用AVX2指令vpgatherdd装载到SIMD寄存器中；2)进入SM4加/解密操作前，依次将32个轮密钥存放在SIMD寄存器中。受限于Intel、AMD处理器中只有16个SIMD寄存器，本文采用第1种方式。实现装载轮密钥的代码如下：

```
_m128i index=_mm_setzero_si128();
R4=_mm_i32gather_epi32((int*)(rk + id), index, 4);
```

轮函数T变换

SM4轮函数中的*T*变换操作是由非线性变换*τ*和线性变换*L*复合而成。并行实现*T*变换有2种策略：（1)分别实现*τ*和*L*；（2)将*τ*和*L*合并实现。

我们下面以策略一为例：*T*变换中的*τ*是由4个S盒操作并置构成。*T*变换中的*τ*是由4个S盒操作并置构成。可将8-bit的S盒输出规模转换为32-bit，借助AVX2的vpgatherdd指令即可实现SM4 S盒操作。我们将8-bit S盒转换为4个8-bit输入32-bit输出表。

将SM4的S盒表记为*ST*，转换后的4个表记为*S*0、*S*1，*S*2、*S*3。*S*0-*S*3的生成规则如下：

$$
\begin{aligned}
&S0\left[i\right]=ST\left[i\right];\\
&S1\left[i\right]=ST\left[i\right]\ll 8;\\
&S2\left[i\right]=ST\left[i\right]\ll 16;\\
&S3\left[i\right]=ST\left[i\right]\ll 24;0\leq i<256.
\end{aligned}
$$

因此，非线性变换*τ*即可通过掩码、移位、异或、查表操作实现。

使用AVX2指令实现如下所示：

```
mask=_mm256_setr_epi32(0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff);
temp1=_mm256_srli_epi32(R, 8);
temp2=_mm256_srli_epi32(R, 16);
temp3=_mm256_srli_epi32(R, 24);
temp1=_mm256_and_si256(temp1, mask);
temp2=_mm256_and_si256(temp2, mask);
temp3=_mm256_and_si256(temp3, mask);
```

*T*变换中的*L*包含循环移位和异或操作，一般来说，循环移位操作可通过左移、右移和异或操作实现。经研究发现，可使用字节置换指令vpshufb优化*L*中包含的4个循环移位操作：

1) <<<24：循环移位24 bit相当于循环移位3个字节位置，这样，可执行vpshufb指令实现；

2) <<<2：分别执行左移指令vpslld、右移指令vpsrld和异或指令vpxor即可实现；

3) <<<10、<<<18：相当于对完成2后再次分别执行<<<8、<<<16。这样，类似于<<<24，<<<8和<<<16可分别执行vpshufb指令实现。

因此，完成线性变换*L*中包含的循环移位操作只需执行1次左移vpslld、1次右移vpsrld、1次异或vpxor、3次字节置换vpshufb共6条AVX2指令即可。

轮函数优化

SM4的轮函数*F*有如下规律：第1、2、3位置的32-bit字*R*1、*R*2、*R*3和轮密钥*R*4经过*T*变换后异或并更新第0位置的32-bit字*R*0，然后4个32-bit字循环左移一个32-bit位置作为下一轮的输入。循环移位32-bit通过调整SIMD寄存器的顺序即可，如轮函数*F*的输入为*R*0、*R*1、*R*2、*R*3，下一轮的输入为*R*1、*R*2、*R*3、*R*0，经过4次轮函数变换后寄存器的位置重新为

*R*0、*R*1、*R*2、*R*3。因此，可以将SM4的4轮变换展开实现，迭代8次即可完成SM4加/解密操作。这样实现轮函数*F*既较少使用SIMD寄存器的个数，并且消除了不同SIMD寄存器内容之间的移动。轮函数*F*优化实现代码如下：

```
for (int i=0; i < 8; ++i)
{
R4 =_mm256_i32gather_epi32((int*)
    (rk + 4 * i + 0), index, 4);
R4 =_mm256_xor_si256(R4, R1);
R4 =_mm256_xor_si256(R4, R2);
R4 =_mm256_xor_si256(R4, R3);
R4 =T(R4);
R0 =_mm256_xor_si256(R0, R4);
R4 =_mm256_i32gather_epi32((int*)
    (rk + 4 * i + 1), index, 4);
R4=_mm256_xor_si256(R4, R0);
R4 =_mm256_xor_si256(R4, R2);
R4 =_mm256_xor_si256(R4, R3);
R4 =T(R4);
R1 =_mm256_xor_si256(R1, R4);
R4 =_mm256_i32gather_epi32((int*)
```

```
    (rk + 4 * i + 2), index, 4);
R4=_mm256_xor_si256(R4, R1);
R4 =_mm256_xor_si256(R4, R0);
R4 =_mm256_xor_si256(R4, R3);
R4 =T(R4);
R2 =_mm256_xor_si256(R2, R4);
R4 =_mm256_i32gather_epi32((int*)
    (rk + 4 * i + 3), index, 4);
R4=_mm256_xor_si256(R4, R2);
R4 =_mm256_xor_si256(R4, R1);
R4 =_mm256_xor_si256(R4, R0);
R4 =T(R4);
R3 =_mm256_xor_si256(R3, R4);
}
```

实验结果

将SIMD与普通实现的结果对比，相比较于查表方法，密码算法使用SIMD指令实现的时候，数据存放在SIMD寄存器中，读取的数据延时较小，更重要的是SIMD指令在并行性方面有明显优势。实验结果如下图，相比于普通方法，SIMD指令有明显的加速效果。

	128-bit异或	8-32查表	4-bit S盒替换
SIMD指令	1.03	7.47	1.05
普通实现	4.03	11.3	24.3

查表优化

基本思路

在SM4加密的轮函数中，设S盒的输入为 $(x0, x1, x2, x3)$，则S盒的输出为 $(S(x0), S(x1), S(x2), S(x3))$。S盒之后是一个基于移位和异或的线性变换函数L。由于S盒的输出可以通过异或拆分为 $(S(x0) << 24) \oplus (S(x1) << 16) \oplus (S(x2) << 8) \oplus S(x3)$，并且L函数由异或进行连接，因此过L函数时 $L(S(x0), S(x1), S(x2), S(x3))$ 可以转化为

$L(S(x0) << 24) \oplus L(S(x1) << 16) \oplus L(S(x2) << 8) \oplus L(S(x0))$。由此可以得出，我们可以构造4个由8-bit串映射到32-bit串的表，其映射关系分别为 $x \rightarrow L(S(x) << 24)$、$x \rightarrow L(S(x) << 16)$、 $x \rightarrow L(S(x) << 8)$ 与 $x \rightarrow L(S(x))$。通过查表然后异或来取代原有的S盒操作和L函数操作将一定程度上节约时间。

实验程序

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include<stdint.h>
```

```cpp
#include<iostream>
#include<cmath>
#include <ctime>
#include <ratio>
#include <chrono>

#define DEBUG
#ifdef DEBUG

//以下为查表优化的四个查找表
static uint32_t Table0[256] = {
    0x8ED55B5B, 0xD0924242, 0x4DEAA7A7, 0x06FDFBFB, 0xFCCF3333, 0x65E28787,
    0xC93DF4F4, 0x6BB5DEDE, 0x4E165858, 0x6EB4DADA, 0x44145050, 0xCAC10B0B,
    0x8828A0A0, 0x17F8EFEF, 0x9C2CB0B0, 0x11051414, 0x872BACAC, 0xFB669D9D,
    0xF2986A6A, 0xAE77D9D9, 0x822AA8A8, 0x46BCFAFA, 0x14041010, 0xCFC00F0F,
    0x02A8AAAA, 0x54451111, 0x5F134C4C, 0xBE269898, 0x6D482525, 0x9E841A1A,
    0x1E061818, 0xFD9B6666, 0xEC9E7272, 0x4A430909, 0x10514141, 0x24F7D3D3,
    0xD5934646, 0x53ECBFBF, 0xF89A6262, 0x927BE9E9, 0xFF33CCCC, 0x04555151,
    0x270B2C2C, 0x4F420D0D, 0x59EEB7B7, 0xF3CC3F3F, 0x1CAEB2B2, 0xEA638989,
    0x74E79393, 0x7FB1CECE, 0x6C1C7070, 0x0DABA6A6, 0xEDCA2727, 0x28082020,
    0x48EBA3A3, 0xC1975656, 0x80820202, 0xA3DC7F7F, 0xC4965252, 0x12F9EBEB,
    0xA174D5D5, 0xB38D3E3E, 0xC33FFCFC, 0x3EA49A9A, 0x5B461D1D, 0x1B071C1C,
    0x3BA59E9E, 0x0CFFF3F3, 0x3FF0CFCF, 0xBF72CDCD, 0x4B175C5C, 0x52B8EAEA,
    0x8F810E0E, 0x3D586565, 0xCC3CF0F0, 0x7D196464, 0x7EE59B9B, 0x91871616,
    0x734E3D3D, 0x08AAA2A2, 0xC869A1A1, 0xC76AADAD, 0x85830606, 0x7AB0CACA,
    0xB570C5C5, 0xF4659191, 0xB2D96B6B, 0xA7892E2E, 0x18FBE3E3, 0x47E8AFAF,
    0x330F3C3C, 0x674A2D2D, 0xB071C1C1, 0x0E575959, 0xE99F7676, 0xE135D4D4,
    0x661E7878, 0xB4249090, 0x360E3838, 0x265F7979, 0xEF628D8D, 0x38596161,
    0x95D24747, 0x2AA08A8A, 0xB1259494, 0xAA228888, 0x8C7DF1F1, 0xD73BECEC,
    0x05010404, 0xA5218484, 0x9879E1E1, 0x9B851E1E, 0x84D75353, 0x00000000,
    0x5E471919, 0x0B565D5D, 0xE39D7E7E, 0x9FD04F4F, 0xBB279C9C, 0x1A534949,
    0x7C4D3131, 0xEE36D8D8, 0x0A020808, 0x7BE49F9F, 0x20A28282, 0xD4C71313,
    0xE8CB2323, 0xE69C7A7A, 0x42E9ABAB, 0x43BDFEFE, 0xA2882A2A, 0x9AD14B4B,
    0x40410101, 0xDBC41F1F, 0xD838E0E0, 0x61B7D6D6, 0x2FA18E8E, 0x2BF4DFDF,
    0x3AF1CBCB, 0xF6CD3B3B, 0x1DFAE7E7, 0xE5608585, 0x41155454, 0x25A38686,
    0x60E38383, 0x16ACBABA, 0x295C7575, 0x34A69292, 0xF7996E6E, 0xE434D0D0,
    0x721A6868, 0x01545555, 0x19AFB6B6, 0xDF914E4E, 0xFA32C8C8, 0xF030C0C0,
    0x21F6D7D7, 0xBC8E3232, 0x75B3C6C6, 0x6FE08F8F, 0x691D7474, 0x2EF5DBDB,
    0x6AE18B8B, 0x962EB8B8, 0x8A800A0A, 0xFE679999, 0xE2C92B2B, 0xE0618181,
    0xC0C30303, 0x8D29A4A4, 0xAF238C8C, 0x07A9AEAE, 0x390D3434, 0x1F524D4D,
    0x764F3939, 0xD36EBDBD, 0x81D65757, 0xB7D86F6F, 0xEB37DCDC, 0x51441515,
    0xA6DD7B7B, 0x09FEF7F7, 0xB68C3A3A, 0x932FBCBC, 0x0F030C0C, 0x03FCFFFF,
    0xC26BA9A9, 0xBA73C9C9, 0xD96CB5B5, 0xDC6DB1B1, 0x375A6D6D, 0x15504545,
    0xB98F3636, 0x771B6C6C, 0x13ADBEBE, 0xDA904A4A, 0x57B9EEEE, 0xA9DE7777,
    0x4CBEF2F2, 0x837EFDFD, 0x55114444, 0xBDDA6767, 0x2C5D7171, 0x45400505,
    0x631F7C7C, 0x50104040, 0x325B6969, 0xB8DB6363, 0x220A2828, 0xC5C20707,
    0xF531C4C4, 0xA88A2222, 0x31A79696, 0xF9CE3737, 0x977AEDED, 0x49BFF6F6,
```

```
    0x992DB4B4, 0xA475D1D1, 0x90D34343, 0x5A124848, 0x58BAE2E2, 0x71E69797,
    0x64B6D2D2, 0x70B2C2C2, 0xAD8B2626, 0xCD68A5A5, 0xCB955E5E, 0x624B2929,
    0x3C0C3030, 0xCE945A5A, 0xAB76DDDD, 0x867FF9F9, 0xF1649595, 0x5DBBE6E6,
    0x35F2C7C7, 0x2D092424, 0xD1C61717, 0xD66FB9B9, 0xDEC51B1B, 0x94861212,
    0x78186060, 0x30F3C3C3, 0x897CF5F5, 0x5CEFB3B3, 0xD23AE8E8, 0xACDF7373,
    0x794C3535, 0xA0208080, 0x9D78E5E5, 0x56EDBBBB, 0x235E7D7D, 0xC63EF8F8,
    0x8BD45F5F, 0xE7C82F2F, 0xDD39E4E4, 0x68492121 };
static uint32_t Table1[256] = {
    0x5B8ED55B, 0x42D09242, 0xA74DEAA7, 0xFB06FDFB, 0x33FCCF33, 0x8765E287,
    0xF4C93DF4, 0xDE6BB5DE, 0x584E1658, 0xDA6EB4DA, 0x50441450, 0x0BCAC10B,
    0xA08828A0, 0xEF17F8EF, 0xB09C2CB0, 0x14110514, 0xAC872BAC, 0x9DFB669D,
    0x6AF2986A, 0xD9AE77D9, 0xA8822AA8, 0xFA46BCFA, 0x10140410, 0x0FCFC00F,
    0xAA02A8AA, 0x11544511, 0x4C5F134C, 0x98BE2698, 0x256D4825, 0x1A9E841A,
    0x181E0618, 0x66FD9B66, 0x72EC9E72, 0x094A4309, 0x41105141, 0xD324F7D3,
    0x46D59346, 0xBF53ECBF, 0x62F89A62, 0xE9927BE9, 0xCCFF33CC, 0x51045551,
    0x2C270B2C, 0x0D4F420D, 0xB759EEB7, 0x3FF3CC3F, 0xB21CAEB2, 0x89EA6389,
    0x9374E793, 0xCE7FB1CE, 0x706C1C70, 0xA60DABA6, 0x27EDCA27, 0x20280820,
    0xA348EBA3, 0x56C19756, 0x02808202, 0x7FA3DC7F, 0x52C49652, 0xEB12F9EB,
    0xD5A174D5, 0x3EB38D3E, 0xFCC33FFC, 0x9A3EA49A, 0x1D5B461D, 0x1C1B071C,
    0x9E3BA59E, 0xF30CFFF3, 0xCF3FF0CF, 0xCDBF72CD, 0x5C4B175C, 0xEA52B8EA,
    0x0E8F810E, 0x653D5865, 0xF0CC3CF0, 0x647D1964, 0x9B7EE59B, 0x16918716,
    0x3D734E3D, 0xA208AAA2, 0xA1C869A1, 0xADC76AAD, 0x06858306, 0xCA7AB0CA,
    0xC5B570C5, 0x91F46591, 0x6BB2D96B, 0x2EA7892E, 0xE318FBE3, 0xAF47E8AF,
    0x3C330F3C, 0x2D674A2D, 0xC1B071C1, 0x590E5759, 0x76E99F76, 0xD4E135D4,
    0x78661E78, 0x90B42490, 0x38360E38, 0x79265F79, 0x8DEF628D, 0x61385961,
    0x4795D247, 0x8A2AA08A, 0x94B12594, 0x88AA2288, 0xF18C7DF1, 0xECD73BEC,
    0x04050104, 0x84A52184, 0xE19879E1, 0x1E9B851E, 0x5384D753, 0x00000000,
    0x195E4719, 0x5D0B565D, 0x7EE39D7E, 0x4F9FD04F, 0x9CBB279C, 0x491A5349,
    0x317C4D31, 0xD8EE36D8, 0x080A0208, 0x9F7BE49F, 0x8220A282, 0x13D4C713,
    0x23E8CB23, 0x7AE69C7A, 0xAB42E9AB, 0xFE43BDFE, 0x2AA2882A, 0x4B9AD14B,
    0x01404101, 0x1FDBC41F, 0xE0D838E0, 0xD661B7D6, 0x8E2FA18E, 0xDF2BF4DF,
    0xCB3AF1CB, 0x3BF6CD3B, 0xE71DFAE7, 0x85E56085, 0x54411554, 0x8625A386,
    0x8360E383, 0xBA16ACBA, 0x75295C75, 0x9234A692, 0x6EF7996E, 0xD0E434D0,
    0x68721A68, 0x55015455, 0xB619AFB6, 0x4EDF914E, 0xC8FA32C8, 0xC0F030C0,
    0xD721F6D7, 0x32BC8E32, 0xC675B3C6, 0x8F6FE08F, 0x74691D74, 0xDB2EF5DB,
    0x8B6AE18B, 0xB8962EB8, 0x0A8A800A, 0x99FE6799, 0x2BE2C92B, 0x81E06181,
    0x03C0C303, 0xA48D29A4, 0x8CAF238C, 0xAE07A9AE, 0x34390D34, 0x4D1F524D,
    0x39764F39, 0xBDD36EBD, 0x5781D657, 0x6FB7D86F, 0xDCEB37DC, 0x15514415,
    0x7BA6DD7B, 0xF709FEF7, 0x3AB68C3A, 0xBC932FBC, 0x0C0F030C, 0xFF03FCFF,
    0xA9C26BA9, 0xC9BA73C9, 0xB5D96CB5, 0xB1DC6DB1, 0x6D375A6D, 0x45155045,
    0x36B98F36, 0x6C771B6C, 0xBE13ADBE, 0x4ADA904A, 0xEE57B9EE, 0x77A9DE77,
    0xF24CBEF2, 0xFD837EFD, 0x44551144, 0x67BDDA67, 0x712C5D71, 0x05454005,
    0x7C631F7C, 0x40501040, 0x69325B69, 0x63B8DB63, 0x28220A28, 0x07C5C207,
    0xC4F531C4, 0x22A88A22, 0x9631A796, 0x37F9CE37, 0xED977AED, 0xF649BFF6,
    0xB4992DB4, 0xD1A475D1, 0x4390D343, 0x485A1248, 0xE258BAE2, 0x9771E697,
    0xD264B6D2, 0xC270B2C2, 0x26AD8B26, 0xA5CD68A5, 0x5ECB955E, 0x29624B29,
    0x303C0C30, 0x5ACE945A, 0xDDAB76DD, 0xF9867FF9, 0x95F16495, 0xE65DBBE6,
```

```
    0xC735F2C7, 0x242D0924, 0x17D1C617, 0xB9D66FB9, 0x1BDEC51B, 0x12948612,
    0x60781860, 0xC330F3C3, 0xF5897CF5, 0xB35CEFB3, 0xE8D23AE8, 0x73ACDF73,
    0x35794C35, 0x80A02080, 0xE59D78E5, 0xBB56EDBB, 0x7D235E7D, 0xF8C63EF8,
    0x5F8BD45F, 0x2FE7C82F, 0xE4DD39E4, 0x21684921 };
static uint32_t Table2[256] = {
    0x5B5B8ED5, 0x4242D092, 0xA7A74DEA, 0xFBFB06FD, 0x3333FCCF, 0x878765E2,
    0xF4F4C93D, 0xDEDE6BB5, 0x58584E16, 0xDADA6EB4, 0x50504414, 0x0B0BCAC1,
    0xA0A08828, 0xEFEF17F8, 0xB0B09C2C, 0x14141105, 0xACAC872B, 0x9D9DFB66,
    0x6A6AF298, 0xD9D9AE77, 0xA8A8822A, 0xFAFA46BC, 0x10101404, 0x0F0FCFC0,
    0xAAAA02A8, 0x11115445, 0x4C4C5F13, 0x9898BE26, 0x25256D48, 0x1A1A9E84,
    0x18181E06, 0x6666FD9B, 0x7272EC9E, 0x09094A43, 0x41411051, 0xD3D324F7,
    0x4646D593, 0xBFBF53EC, 0x6262F89A, 0xE9E9927B, 0xCCCCFF33, 0x51510455,
    0x2C2C270B, 0x0D0D4F42, 0xB7B759EE, 0x3F3FF3CC, 0xB2B21CAE, 0x8989EA63,
    0x939374E7, 0xCECE7FB1, 0x70706C1C, 0xA6A60DAB, 0x2727EDCA, 0x20202808,
    0xA3A348EB, 0x5656C197, 0x02028082, 0x7F7FA3DC, 0x5252C496, 0xEBEB12F9,
    0xD5D5A174, 0x3E3EB38D, 0xFCFCC33F, 0x9A9A3EA4, 0x1D1D5B46, 0x1C1C1B07,
    0x9E9E3BA5, 0xF3F30CFF, 0xCFCF3FF0, 0xCDCDBF72, 0x5C5C4B17, 0xEAEA52B8,
    0x0E0E8F81, 0x65653D58, 0xF0F0CC3C, 0x64647D19, 0x9B9B7EE5, 0x16169187,
    0x3D3D734E, 0xA2A208AA, 0xA1A1C869, 0xADADC76A, 0x06068583, 0xCACA7AB0,
    0xC5C5B570, 0x9191F465, 0x6B6BB2D9, 0x2E2EA789, 0xE3E318FB, 0xAFAF47E8,
    0x3C3C330F, 0x2D2D674A, 0xC1C1B071, 0x59590E57, 0x7676E99F, 0xD4D4E135,
    0x7878661E, 0x9090B424, 0x3838360E, 0x7979265F, 0x8D8DEF62, 0x61613859,
    0x474795D2, 0x8A8A2AA0, 0x9494B125, 0x8888AA22, 0xF1F18C7D, 0xECECD73B,
    0x04040501, 0x8484A521, 0xE1E19879, 0x1E1E9B85, 0x535384D7, 0x00000000,
    0x19195E47, 0x5D5D0B56, 0x7E7EE39D, 0x4F4F9FD0, 0x9C9CBB27, 0x49491A53,
    0x31317C4D, 0xD8D8EE36, 0x08080A02, 0x9F9F7BE4, 0x828220A2, 0x1313D4C7,
    0x2323E8CB, 0x7A7AE69C, 0xABAB42E9, 0xFEFE43BD, 0x2A2AA288, 0x4B4B9AD1,
    0x01014041, 0x1F1FDBC4, 0xE0E0D838, 0xD6D661B7, 0x8E8E2FA1, 0xDFDF2BF4,
    0xCBCB3AF1, 0x3B3BF6CD, 0xE7E71DFA, 0x8585E560, 0x54544115, 0x868625A3,
    0x838360E3, 0xBABA16AC, 0x7575295C, 0x929234A6, 0x6E6EF799, 0xD0D0E434,
    0x6868721A, 0x55550154, 0xB6B619AF, 0x4E4EDF91, 0xC8C8FA32, 0xC0C0F030,
    0xD7D721F6, 0x3232BC8E, 0xC6C675B3, 0x8F8F6FE0, 0x7474691D, 0xDBDB2EF5,
    0x8B8B6AE1, 0xB8B8962E, 0x0A0A8A80, 0x9999FE67, 0x2B2BE2C9, 0x8181E061,
    0x0303C0C3, 0xA4A48D29, 0x8C8CAF23, 0xAEAE07A9, 0x3434390D, 0x4D4D1F52,
    0x3939764F, 0xBDBDD36E, 0x575781D6, 0x6F6FB7D8, 0xDCDCEB37, 0x15155144,
    0x7B7BA6DD, 0xF7F709FE, 0x3A3AB68C, 0xBCBC932F, 0x0C0C0F03, 0xFFFF03FC,
    0xA9A9C26B, 0xC9C9BA73, 0xB5B5D96C, 0xB1B1DC6D, 0x6D6D375A, 0x45451550,
    0x3636B98F, 0x6C6C771B, 0xBEBE13AD, 0x4A4ADA90, 0xEEEE57B9, 0x7777A9DE,
    0xF2F24CBE, 0xFDFD837E, 0x44445511, 0x6767BDDA, 0x71712C5D, 0x05054540,
    0x7C7C631F, 0x40405010, 0x6969325B, 0x6363B8DB, 0x2828220A, 0x0707C5C2,
    0xC4C4F531, 0x2222A88A, 0x969631A7, 0x3737F9CE, 0xEDED977A, 0xF6F649BF,
    0xB4B4992D, 0xD1D1A475, 0x434390D3, 0x48485A12, 0xE2E258BA, 0x979771E6,
    0xD2D264B6, 0xC2C270B2, 0x2626AD8B, 0xA5A5CD68, 0x5E5ECB95, 0x2929624B,
    0x30303C0C, 0x5A5ACE94, 0xDDDDAB76, 0xF9F9867F, 0x9595F164, 0xE6E65DBB,
    0xC7C735F2, 0x24242D09, 0x1717D1C6, 0xB9B9D66F, 0x1B1BDEC5, 0x12129486,
    0x60607818, 0xC3C330F3, 0xF5F5897C, 0xB3B35CEF, 0xE8E8D23A, 0x7373ACDF,
    0x3535794C, 0x8080A020, 0xE5E59D78, 0xBBBB56ED, 0x7D7D235E, 0xF8F8C63E,
```

```
    0x5F5F8BD4, 0x2F2FE7C8, 0xE4E4DD39, 0x21216849 };
static uint32_t Table3[256] = {
    0xD55B5B8E, 0x924242D0, 0xEAA7A74D, 0xFDFBFB06, 0xCF3333FC, 0xE2878765,
    0x3DF4F4C9, 0xB5DEDE6B, 0x1658584E, 0xB4DADA6E, 0x14505044, 0xC10B0BCA,
    0x28A0A088, 0xF8EFEF17, 0x2CB0B09C, 0x05141411, 0x2BACAC87, 0x669D9DFB,
    0x986A6AF2, 0x77D9D9AE, 0x2AA8A882, 0xBCFAFA46, 0x04101014, 0xC00F0FCF,
    0xA8AAAA02, 0x45111154, 0x134C4C5F, 0x269898BE, 0x4825256D, 0x841A1A9E,
    0x0618181E, 0x9B6666FD, 0x9E7272EC, 0x4309094A, 0x51414110, 0xF7D3D324,
    0x934646D5, 0xECBFBF53, 0x9A6262F8, 0x7BE9E992, 0x33CCCCFF, 0x55515104,
    0x0B2C2C27, 0x420D0D4F, 0xEEB7B759, 0xCC3F3FF3, 0xAEB2B21C, 0x638989EA,
    0xE7939374, 0xB1CECE7F, 0x1C70706C, 0xABA6A60D, 0xCA2727ED, 0x08202028,
    0xEBA3A348, 0x975656C1, 0x82020280, 0xDC7F7FA3, 0x965252C4, 0xF9EBEB12,
    0x74D5D5A1, 0x8D3E3EB3, 0x3FFCFCC3, 0xA49A9A3E, 0x461D1D5B, 0x071C1C1B,
    0xA59E9E3B, 0xFFF3F30C, 0xF0CFCF3F, 0x72CDCDBF, 0x175C5C4B, 0xB8EAEA52,
    0x810E0E8F, 0x5865653D, 0x3CF0F0CC, 0x1964647D, 0xE59B9B7E, 0x87161691,
    0x4E3D3D73, 0xAAA2A208, 0x69A1A1C8, 0x6AADADC7, 0x83060685, 0xB0CACA7A,
    0x70C5C5B5, 0x659191F4, 0xD96B6BB2, 0x892E2EA7, 0xFBE3E318, 0xE8AFAF47,
    0x0F3C3C33, 0x4A2D2D67, 0x71C1C1B0, 0x5759590E, 0x9F7676E9, 0x35D4D4E1,
    0x1E787866, 0x249090B4, 0x0E383836, 0x5F797926, 0x628D8DEF, 0x59616138,
    0xD2474795, 0xA08A8A2A, 0x259494B1, 0x228888AA, 0x7DF1F18C, 0x3BECECD7,
    0x01040405, 0x218484A5, 0x79E1E198, 0x851E1E9B, 0xD7535384, 0x00000000,
    0x4719195E, 0x565D5D0B, 0x9D7E7EE3, 0xD04F4F9F, 0x279C9CBB, 0x5349491A,
    0x4D31317C, 0x36D8D8EE, 0x0208080A, 0xE49F9F7B, 0xA2828220, 0xC71313D4,
    0xCB2323E8, 0x9C7A7AE6, 0xE9ABABA42, 0xBDFEFE43, 0x882A2AA2, 0xD14B4B9A,
    0x41010140, 0xC41F1FDB, 0x38E0E0D8, 0xB7D6D661, 0xA18E8E2F, 0xF4DFDF2B,
    0xF1CBCB3A, 0xCD3B3BF6, 0xFAE7E71D, 0x608585E5, 0x15545441, 0xA3868625,
    0xE3838360, 0xACBABA16, 0x5C757529, 0xA6929234, 0x996E6EF7, 0x34D0D0E4,
    0x1A686872, 0x54555501, 0xAFB6B619, 0x914E4EDF, 0x32C8C8FA, 0x30C0C0F0,
    0xF6D7D721, 0x8E3232BC, 0xB3C6C675, 0xE08F8F6F, 0x1D747469, 0xF5DBDB2E,
    0xE18B8B6A, 0x2EB8B896, 0x800A0A8A, 0x679999FE, 0xC92B2BE2, 0x618181E0,
    0xC30303C0, 0x29A4A48D, 0x238C8CAF, 0xA9AEAE07, 0x0D343439, 0x524D4D1F,
    0x4F393976, 0x6EBDBDD3, 0xD6575781, 0xD86F6FB7, 0x37DCDCEB, 0x44151551,
    0xDD7B7BA6, 0xFEF7F709, 0x8C3A3AB6, 0x2FBCBC93, 0x030C0C0F, 0xFCFFFF03,
    0x6BA9A9C2, 0x73C9C9BA, 0x6CB5B5D9, 0x6DB1B1DC, 0x5A6D6D37, 0x50454515,
    0x8F3636B9, 0x1B6C6C77, 0xADBEBE13, 0x904A4ADA, 0xB9EEEE57, 0xDE7777A9,
    0xBEF2F24C, 0x7EFDFD83, 0x11444455, 0xDA6767BD, 0x5D71712C, 0x40050545,
    0x1F7C7C63, 0x10404050, 0x5B696932, 0xDB6363B8, 0x0A282822, 0xC20707C5,
    0x31C4C4F5, 0x8A2222A8, 0xA7969631, 0xCE3737F9, 0x7AEDED97, 0xBFF6F649,
    0x2DB4B499, 0x75D1D1A4, 0xD3434390, 0x1248485A, 0xBAE2E258, 0xE6979771,
    0xB6D2D264, 0xB2C2C270, 0x8B2626AD, 0x68A5A5CD, 0x955E5ECB, 0x4B292962,
    0x0C30303C, 0x945A5ACE, 0x76DDDDAB, 0x7FF9F986, 0x649595F1, 0xBBE6E65D,
    0xF2C7C735, 0x0924242D, 0xC61717D1, 0x6FB9B9D6, 0xC51B1BDE, 0x86121294,
    0x18606078, 0xF3C3C330, 0x7CF5F589, 0xEFB3B35C, 0x3AE8E8D2, 0xDF7373AC,
    0x4C353579, 0x208080A0, 0x78E5E59D, 0xEDBBBB56, 0x5E7D7D23, 0x3EF8F8C6,
    0xD45F5F8B, 0xC82F2FE7, 0x39E4E4DD, 0x49212168 };

uint8_t S_box[16][16] =
```

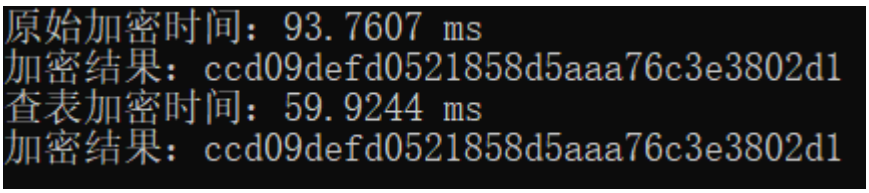
```
{
        {0xd6,0x90,0xe9,0xfe,0xcc,0xe1,0x3d,0xb7,0x16,0xb6,0x14,0xc2,0x28,0xfb,0
        {0x2b,0x67,0x9a,0x76,0x2a,0xbe,0x04,0xc3,0xaa,0x44,0x13,0x26,0x49,0x86,0
        {0x9c,0x42,0x50,0xf4,0x91,0xef,0x98,0x7a,0x33,0x54,0x0b,0x43,0xed,0xcf,0
        {0xe4,0xb3,0x1c,0xa9,0xc9,0x08,0xe8,0x95,0x80,0xdf,0x94,0xfa,0x75,0x8f,0
        {0x47,0x07,0xa7,0xfc,0xf3,0x73,0x17,0xba,0x83,0x59,0x3c,0x19,0xe6,0x85,0
        {0x68,0x6b,0x81,0xb2,0x71,0x64,0xda,0x8b,0xf8,0xeb,0x0f,0x4b,0x70,0x56,0
        {0x1e,0x24,0x0e,0x5e,0x63,0x58,0xd1,0xa2,0x25,0x22,0x7c,0x3b,0x01,0x21,0
        {0xd4,0x00,0x46,0x57,0x9f,0xd3,0x27,0x52,0x4c,0x36,0x02,0xe7,0xa0,0xc4,0
        {0xea,0xbf,0x8a,0xd2,0x40,0xc7,0x38,0xb5,0xa3,0xf7,0xf2,0xce,0xf9,0x61,0
        {0xe0,0xae,0x5d,0xa4,0x9b,0x34,0x1a,0x55,0xad,0x93,0x32,0x30,0xf5,0x8c,0
        {0x1d,0xf6,0xe2,0x2e,0x82,0x66,0xca,0x60,0xc0,0x29,0x23,0xab,0x0d,0x53,0
        {0xd5,0xdb,0x37,0x45,0xde,0xfd,0x8e,0x2f,0x03,0xff,0x6a,0x72,0x6d,0x6c,0
        {0x8d,0x1b,0xaf,0x92,0xbb,0xdd,0xbc,0x7f,0x11,0xd9,0x5c,0x41,0x1f,0x10,0
        {0x0a,0xc1,0x31,0x88,0xa5,0xcd,0x7b,0xbd,0x2d,0x74,0xd0,0x12,0xb8,0xe5,0
        {0x89,0x69,0x97,0x4a,0x0c,0x96,0x77,0x7e,0x65,0xb9,0xf1,0x09,0xc5,0x6e,0
        {0x18,0xf0,0x7d,0xec,0x3a,0xdc,0x4d,0x20,0x79,0xee,0x5f,0x3e,0xd7,0xcb,0
};
uint32_t FK[4] = { 0xa3b1bac6,0x56aa3350,0x677d9197,0xb27022dc };
uint32_t CK[32] =
{
        0x00070e15,0x1c232a31,0x383f464d,0x545b6269,
        0x70777e85,0x8c939aa1,0xa8afb6bd,0xc4cbd2d9,
        0xe0e7eef5,0xfc030a11,0x181f262d,0x343b4249,
        0x50575e65,0x6c737a81,0x888f969d,0xa4abb2b9,
        0xc0c7ced5,0xdce3eaf1,0xf8ff060d,0x141b2229,
        0x30373e45,0x4c535a61,0x686f767d,0x848b9299,
        0xa0a7aeb5,0xbcc3cad1,0xd8dfe6ed,0xf4fb0209,
        0x10171e25,0x2c333a41,0x484f565d,0x646b7279
};

uint32_t gen_key(uint32_t k0, uint32_t k1, uint32_t k2, uint32_t k3, uint32_t ck
{
    uint32_t mid = k1 ^ k2 ^ k3 ^ ck;
    uint8_t mid0, mid1, mid2, mid3;
    mid0 = mid >> 24;
    mid1 = mid >> 16;
    mid2 = mid >> 8;
    mid3 = mid;
    mid0 = S_box[mid0 >> 4][mid0 & 0b1111];
    mid1 = S_box[mid1 >> 4][mid1 & 0b1111];
    mid2 = S_box[mid2 >> 4][mid2 & 0b1111];
    mid3 = S_box[mid3 >> 4][mid3 & 0b1111];
    uint32_t SR = (mid0 << 24) ^ (mid1 << 16) ^ (mid2 << 8) ^ mid3;
    uint32_t SR23 = (SR << 23) ^ (SR >> 9);
    uint32_t SR13 = (SR << 13) ^ (SR >> 19);
    uint32_t RE = SR ^ SR13 ^ SR23 ^ k0;
```

```
240     return RE;
241 }
242 
243 /*原始版本的轮函数*/
244 void round_fun(uint32_t* x, uint32_t k) {
245         uint32_t A = x[1] ^ x[2] ^ x[3] ^ k;
246     uint8_t A_chr0, A_chr1, A_chr2, A_chr3;
247     A_chr0 = A >> 24;
248     A_chr1 = A >> 16;
249     A_chr2 = A >> 8;
250     A_chr3 = A;
251     A_chr0 = S_box[(A_chr0 >> 4)][A_chr0 & 0b1111];
252     A_chr1 = S_box[(A_chr1 >> 4)][A_chr1 & 0b1111];
253     A_chr2 = S_box[(A_chr2 >> 4)][A_chr2 & 0b1111];
254     A_chr3 = S_box[(A_chr3 >> 4)][A_chr3 & 0b1111];
255     uint32_t B = (A_chr0 << 24) ^ (A_chr1 << 16) ^ (A_chr2 << 8) ^ A_chr3;
256     uint32_t B2 = (B << 2) ^ (B >> 30);
257     uint32_t B10 = (B << 10) ^ (B >> 22);
258     uint32_t B18 = (B << 18) ^ (B >> 14);
259     uint32_t B24 = (B << 24) ^ (B >> 8);
260     uint32_t temp = x[0] ^ B ^ B2 ^ B10 ^ B18 ^ B24;
261         x[0] = x[1];
262         x[1] = x[2];
263         x[2] = x[3];
264         x[3] = temp;
265 }
266 
267 /*查表版本的轮函数*/
268 void round_fun_cklst(uint32_t* x, uint32_t k) {
269     uint32_t A = x[1] ^ x[2] ^ x[3] ^ k;
270     uint8_t A_chr0,A_chr1, A_chr2, A_chr3;
271     A_chr0 = A >> 24;
272     A_chr1 = A >> 16;
273     A_chr2 = A >> 8;
274     A_chr3 = A;
275     uint32_t B;
276     B = Table0[A_chr0] ^ Table1[A_chr1] ^ Table2[A_chr2] ^ Table3[A_chr3] ^ x[0]
277     x[0] = x[1];
278     x[1] = x[2];
279     x[2] = x[3];
280     x[3] = B;
281 }
282 
283 void get_rnd_key(uint32_t* key,uint32_t* roundkey) {
284     roundkey[0] = FK[0] ^ key[0];
285     roundkey[1] = FK[1] ^ key[1];
286     roundkey[2] = FK[2] ^ key[2];
```

```
287     roundkey[3] = FK[3] ^ key[3];
288         for (int i = 4; i < 36; i++) {
289                 roundkey[i] = gen_key(roundkey[i - 4], roundkey[i - 3], roundkey
290         }
291 }
292
293 int main() {
294
295     uint32_t input[4] = { 0x011111111,0xffffffff,0x88888888,0x55555555 };
296         uint32_t output[4] = { 0x0,0x0,0x0,0x0 };
297         uint32_t key[4] = { 0x01234567,0x89abcdef,0xfedcba98,0x76543210 };
298     uint32_t roundkey[36];
299     get_rnd_key(key, roundkey);
300     auto start = std::chrono::high_resolution_clock::now();
301     //为了让运行时间更稳定，此处将32轮的加密进行100000次
302     for (int i = 0; i < 100000; i++) {
303         for (int i_0 = 0; i_0 < 32; i_0++) {
304             round_fun(input, roundkey[i_0 + 4]);
305         };
306     }
307     output[0] = input[3];
308     output[1] = input[2];
309     output[2] = input[1];
310     output[3] = input[0];
311     auto end = std::chrono::high_resolution_clock::now();
312     std::chrono::duration<double, std::ratio<1, 1000>> diff = end - start;
313     std::cout << "原始加密时间: " << diff.count() << " ms\n";
314         printf("加密结果: %02x%02x%02x%02x\n", output[0], output[1], output[2], ou
315
316
317     uint32_t input2[4] = { 0x011111111,0xffffffff,0x88888888,0x55555555 };
318     uint32_t output2[4] = { 0x0,0x0,0x0,0x0 };
319     uint32_t key2[4] = { 0x01234567,0x89abcdef,0xfedcba98,0x76543210 };
320     uint32_t roundkey2[36];
321     get_rnd_key(key2, roundkey2);
322     auto start2 = std::chrono::high_resolution_clock::now();
323     //为了让运行时间更稳定，此处将32轮的加密进行100000次
324     for (int i = 0; i < 100000; i++) {
325         for (int i_0 = 0; i_0 < 32; i_0++) {
326             round_fun_cklst(input2, roundkey2[i_0 + 4]);
327         };
328     }
329     output2[0] = input2[3];
330     output2[1] = input2[2];
331     output2[2] = input2[1];
332     output2[3] = input2[0];
333     auto end2 = std::chrono::high_resolution_clock::now();
```

```
    std::chrono::duration<double, std::ratio<1, 1000>> diff2 = end2 - start2;
    std::cout << "查表加密时间: " << diff2.count() << " ms\n";
    printf("加密结果: %02x%02x%02x%02x\n", output2[0], output2[1], output2[2], ou
    return 0;
}

#endif // DEBUG
```

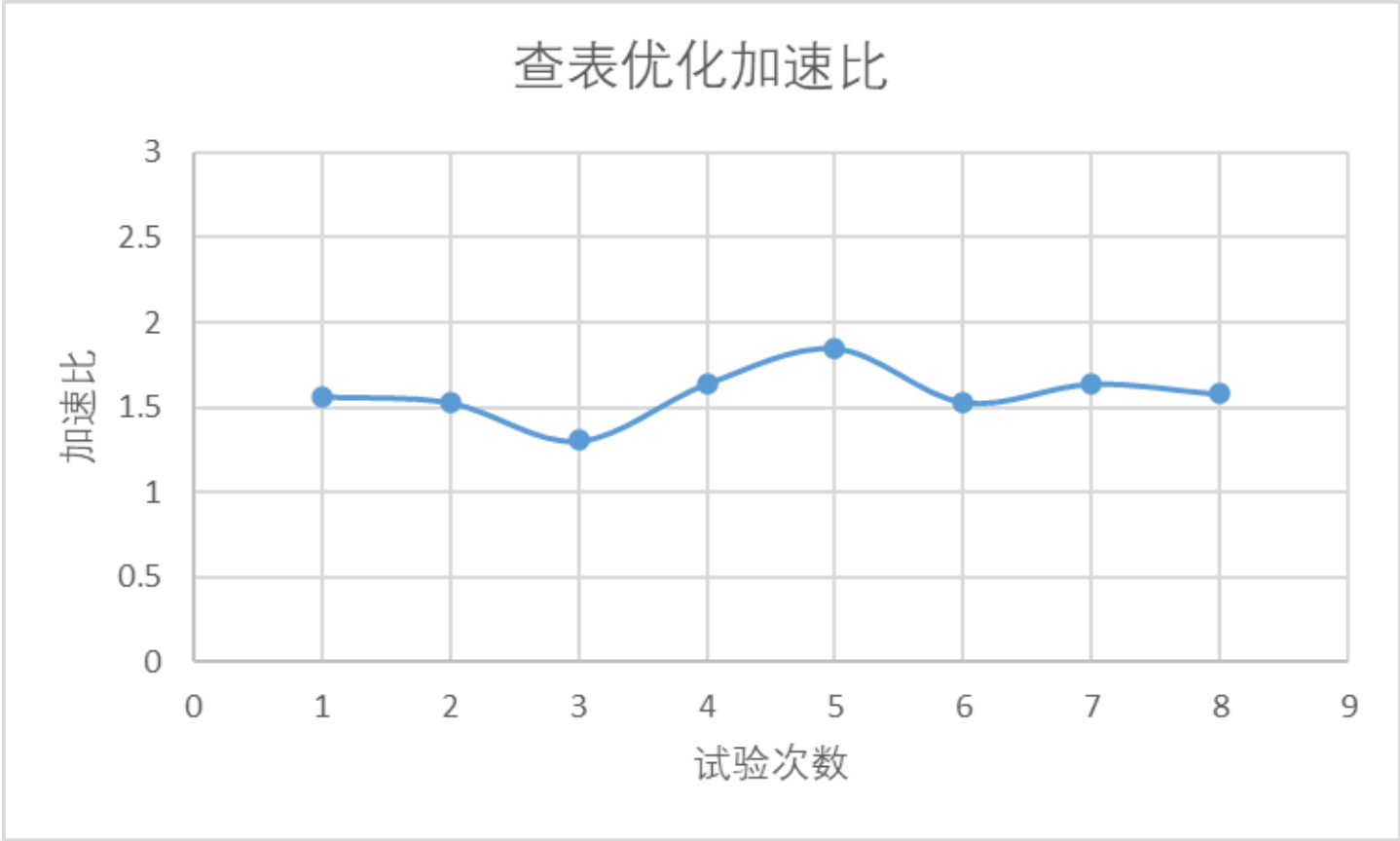
运行结果



性能分析

可见查表优化后的加密结果与原始版本的加密结果相同，验证了查表优化的正确性。在时间上，我们反复测量了8次，得到的数据如下：

原始时间(ms)	93.7607	93.1745	100.225	96.0688	109.904	91.7095	95.9
查表时间(ms)	59.9244	60.8943	76.8167	58.5665	59.4657	59.8373	58.4
加速比	1.56465	1.530102	1.304729	1.640337	1.848191	1.532648	1.64



可以看出本次实验中单纯使用查表优化的加速比基本稳定在1.5左右。

循环展开

基本思路

注意到base程序中直接在m上覆写数据，比较慢，而且每四轮都会回归一次，所以我们可以试着将四轮函数展开，做八次四轮展开。

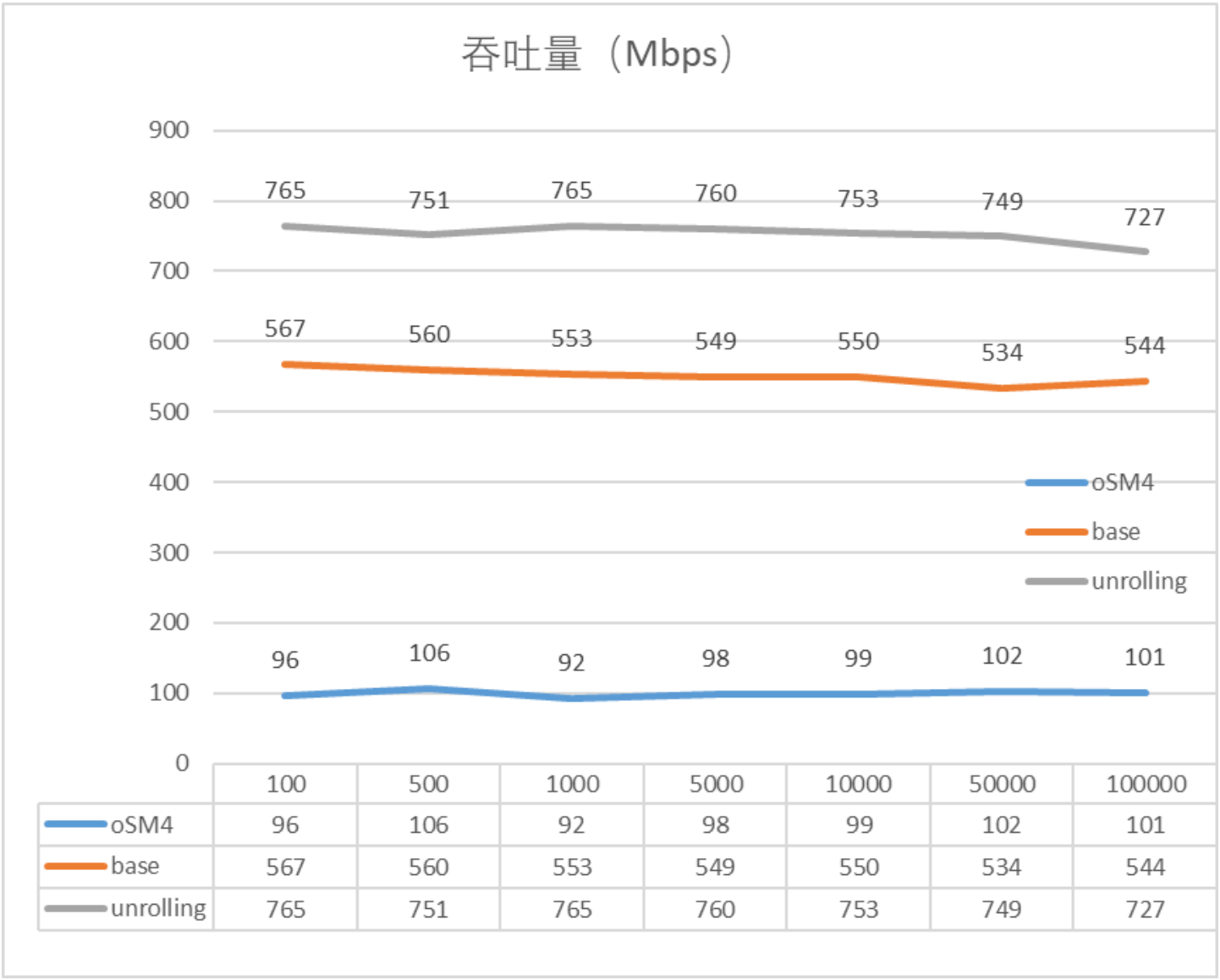
实验程序

```
void enc1(u32 m[],u32* rk)
{

    register u32 m0=m[0],m1=m[1],m2=m[2],m3=m[3];

    for(int i=0;i<32;i+=4)
    {
        register u32 tmp;
        tmp=m1^m2^m3^*rk;
        tmp=(Sbox[(tmp>>24)&0xff]<<24)|
            (Sbox[(tmp>>16)&0xff]<<16)|
            (Sbox[(tmp>>8)&0xff]<<8)|
            (Sbox[tmp&0xff]);
        rk++;
        m0=tmp^rot(tmp,32,2)^rot(tmp,32,10)^rot(tmp,32,18)^rot(tmp,32,24)^m0;
```

```
        tmp=m0^m2^m3^*rk;
        tmp=(Sbox[(tmp>>24)&0xff]<<24)|
            (Sbox[(tmp>>16)&0xff]<<16)|
            (Sbox[(tmp>>8)&0xff]<<8)|
            (Sbox[tmp&0xff]);
        rk++;
        m1=tmp^rot(tmp,32,2)^rot(tmp,32,10)^rot(tmp,32,18)^rot(tmp,32,24)^m1;

        tmp=m1^m0^m3^*rk;
        tmp=(Sbox[(tmp>>24)&0xff]<<24)|
            (Sbox[(tmp>>16)&0xff]<<16)|
            (Sbox[(tmp>>8)&0xff]<<8)|
            (Sbox[tmp&0xff]);
        rk++;
        m2=tmp^rot(tmp,32,2)^rot(tmp,32,10)^rot(tmp,32,18)^rot(tmp,32,24)^m2;
        tmp=m1^m2^m0^*rk;
        tmp=(Sbox[(tmp>>24)&0xff]<<24)|
            (Sbox[(tmp>>16)&0xff]<<16)|
            (Sbox[(tmp>>8)&0xff]<<8)|
            (Sbox[tmp&0xff]);
        rk++;
        m3=tmp^rot(tmp,32,2)^rot(tmp,32,10)^rot(tmp,32,18)^rot(tmp,32,24)^m3;
    }
    m[3]=m0,m[2]=m1,m[1]=m2,m[0]=m3;
}
```

实验效果

同前文，制作了吞吐量关于数据规模的图，发现相比于原初版本提升了七倍，相比于不展开的版本提升了40%。



	100	500	1000	5000	10000	50000	100000
oSM4	96	106	92	98	99	102	101
base	567	560	553	549	550	534	544
unrolling	765	751	765	760	753	749	727

bitslice

本部分由王超然负责。

原理简介

bitslice，即位切片，通过将数据切片为多组数据进行寄存器内的并行，理想状态下能充分利用所有寄存器的效能，并通过切断数据依赖达到充分利用流水线的目的。

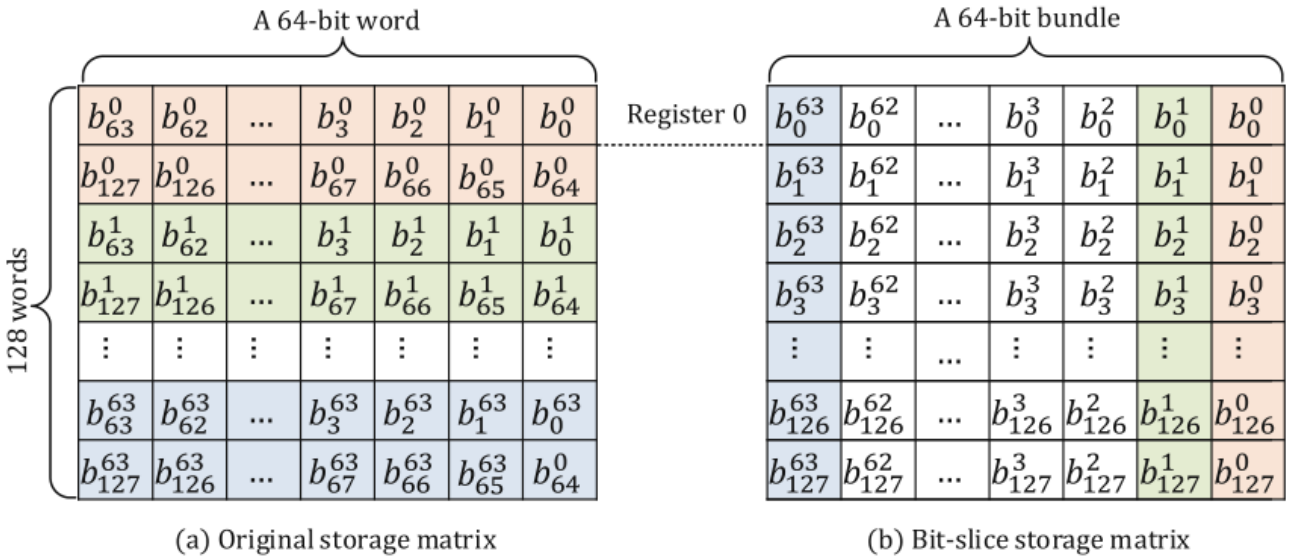


Fig. 2. Transformation from the original storage form to the bit-slice storage form on the 64-bit machine.

（图源Fast Implementation for SM4 Cipher Algorithm Based on Bit-Slice Technology，2018）

通过这种方式，我们将输入划分为尽可能小的独立单元，然后进行划分。根据GmSSL的公告，他们在早年实现了一个bs的版本，本着不重复造复杂轮子的原则，我们不再重构代码，仅在本地做复现。

理论分析

在前文分析中，因为我们没有良好的SIMD查表指令，查表这一步成为了瓶颈；但是在bitslice中，我们通过数学变换直接进行了Sbox的运算，具体原理如下：

SM4的构造虽然有较大的具体差异，但都是线性运算和求逆运算的结合，其中求逆运算是核心功能。

求逆可以转化为在对应域上的幂运算，但直接在GF(2^8)上的运算很麻烦，而另一方面，二次扩域的求逆较为简单，所以我们可以把他转化为多个二次扩域的复合进行计算。

域复合

在 $GF(p^n)$ 的空间里寻求一个 $gcd(r,n)!=1$ ，使得 $GF(p^r)$ 是原空间的一个子空间，则按照子空间进行划分，我们就能划分出商空间 $GF(p^n/p^r)$ 使得 x^r 为最小未定元单位。

使用分治，我们可以通过两次划分获得 $GF(2^8)$ 与 $GF(2^2)$ 产生同构，即 $GF(2^8/2^4),GF(2^4/2^2)$ ，在这几组子域的二次扩域均为两组基。

求逆

由于求逆主要涉及幂运算，在正规基下乘法运算特别是幂运算更加方便，所以取正规基下的求逆过程。

模元r我们可以表示为 $r(y)=y^2+by+c=(y+Y^{p^r})(y+Y)$,即 $b=Y^{p^r}+Y$ ， $c=Y^{p^r}\cdot Y$

我们可以取任意元素g，使得 $g=g_1Y^{p^r}+g_0Y$,那么其逆元 $d=d_1Y^{p^r}+d_0Y$ 满足

$$gd=g_1d_1(Y^{p^r})^2+(g_1d_0+g_0d_1)Y^{p^r}Y+d_0d_1Y^2$$

$= g_1 d_1 (bY^{p^r} + c) + (g_1 d_0 + g_0 d_1)c + g_0 d_0 (bY + c)$

$= (g_1 d_1 bY^{p^r} + g_0 d_0 bY) + [g_1 d_1 c + (g_1 d_0 + g_0 d_1)c + g_0 d_0 c]$

$= (g_1 d_1 bY^{p^r} + g_0 d_0 bY) + [(g_1 + g_0)(d_1 + d_0)c] b^{-1} (Y^{p^r} + Y)$

$= [g_1 d_1 b + (g_1 + g_0)(d_1 + d_0)cb^{-1}] Y^{p^r} + [g_0 d_0 b + (g_1 + g_0)(d_1 + d_0)cb^{-1}] Y)$

$= 1 = b^{-1}(Y^{p^r} + Y)$

得

$b^{-1} = g_1 d_1 b + (g_1 + g_0)(d_1 + d_0)cb^{-1}$

$b^{-1} = g_0 d_0 b + (g_1 + g_0)(d_1 + d_0)cb^{-1}$

相加有

$g_0 d_0 + g_1 d_1 = 0$，即$g_0 d_0 = g_1 d_1$

乘b得

$1 = b^2 g_0 d_0 + (g_1 d_0 + g_0 d_1)c$

乘 g_1 得

$g_1 = g_0 g_1 d_0 b^2 + (g_1^2 d_0 + g_0 g_1 d_1)c$

$= g_0 g_1 d_0 b^2 + (g_1^2 d_0 + g_0^2 d_0)c$

$= [g_0 g_1 b^2 + (g_1^2 + g_0^2)c] d_0$

即

$d_0 = [g_0 g_1 b^2 + (g_1^2 + g_0^2)c]^{-1} g_1$

$d_1 = [g_0 g_1 b^2 + (g_1^2 + g_0^2)c]^{-1} g_0$

至此便求出了逆元的表达式。

而在 $GF(2^2)$ 上

r(y)有且仅有 $r(y) = y^2 + y + 1$,即b=c=1

在这里有 $g_i^n = g_i$

则

$d_0 = [g_0 g_1 + (g_1^2 + g_0^2)]^{-1} g_1$

$= [g_0 g_1 + (g_1^2 + g_0^2)] g_1$

$= g_0 g_1^2 + g_1^3 + g_0^2 g_1$

$= g_0 g_1 + g_1 + g_0 g_1$

$= g_1$

同理

$d_1 = g_0$

这样就完成了递归求逆。

基变换

在应用中，b常取1，C取前一组基。

通过三次扩域计算根据模元将原式转换为3个两组基，再对应相乘即可得到八组基，与原式一一对应。

即

$$
\begin{aligned}
& g_7A^7+g_6A^6+g_5A^5+g_4A^4+g_3A^3+g_2A^2+g_1A+g_0 \\
= & b_7w^2z^4y^{16}+b_6wz^4y^{16}+b_5w^2zy^{16}+b_4wzy^{16}+b_3w^2z^4y+b_2wz^4y+b_1w^2zy+b_0wzy
\end{aligned}
$$

通过以上过程求出y,z,w三组基变换矩阵和逆变换矩阵。与对应的矩阵乘起来即可。

实现

可以发现第一次划分自然而然的把过Sbox的值分成了高4bits和低4bits，因此我们使用4bits作为最小单位；最大普通寄存器为64位，所以我们可以同时16路进行并行；GmSSL也是这么实现的。

实验效果

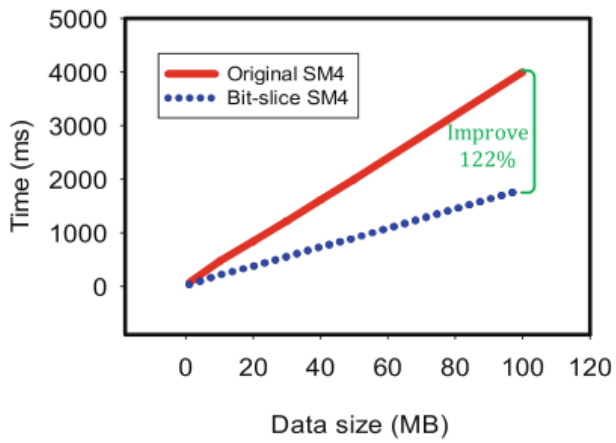
将文件进行gcc编译，发现报错，原因是在C中使用了Cpp特有的引用导致的。

换用g++进行编译及运行：

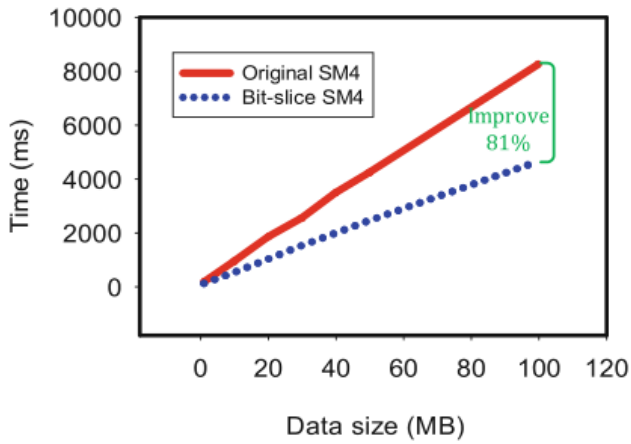
```
┌─[zuni-w@ubuntu:~/Desktop/csapp/mylab3]─[02:54:19 PM]
└─>$ ./test_bs
pai:
100000
 8.158681 s,23.939225 Mb/s,3137762.096594 B/s
44,f6,8e,0e,1e,d8,21,8d,f7,d9,55,07,3d,ba,09,7e,
44,f6,8e,0e,1e,d8,21,8d,f7,d9,55,07,3d,ba,09,7e,
44,f6,8e,0e,1e,d8,21,8d,f7,d9,55,07,3d,ba,09,7e,
44,f6,8e,0e,1e,d8,21,8d,f7,d9,55,07,3d,ba,09,7e,
44,f6,8e,0e,1e,d8,21,8d,f7,d9,55,07,3d,ba,09,7e,
44,f6,8e,0e,1e,d8,21,8d,f7,d9,55,07,3d,ba,09,7e,
44,f6,8e,0e,1e,d8,21,8d,f7,d9,55,07,3d,ba,09,7e,
44,f6,8e,0e,1e,d8,21,8d,f7,d9,55,07,3d,ba,09,7e,
44,f6,8e,0e,1e,d8,21,8d,f7,d9,55,07,3d,ba,09,7e,
44,f6,8e,0e,1e,d8,21,8d,f7,d9,55,07,3d,ba,09,7e,
44,f6,8e,0e,1e,d8,21,8d,f7,d9,55,07,3d,ba,09,7e,
44,f6,8e,0e,1e,d8,21,8d,f7,d9,55,07,3d,ba,09,7e,
44,f6,8e,0e,1e,d8,21,8d,f7,d9,55,07,3d,ba,09,7e,
44,f6,8e,0e,1e,d8,21,8d,f7,d9,55,07,3d,ba,09,7e,
44,f6,8e,0e,1e,d8,21,8d,f7,d9,55,07,3d,ba,09,7e,
44,f6,8e,0e,1e,d8,21,8d,f7,d9,55,07,3d,ba,09,7e,
┌─[zuni-w@ubuntu:~/Desktop/csapp/mylab3]─[02:54:40 PM]
└─>$
```

大大出人意料，这个并行速度甚至为原初版本的1/5；

于是我们分析了原论文中的实验数据：

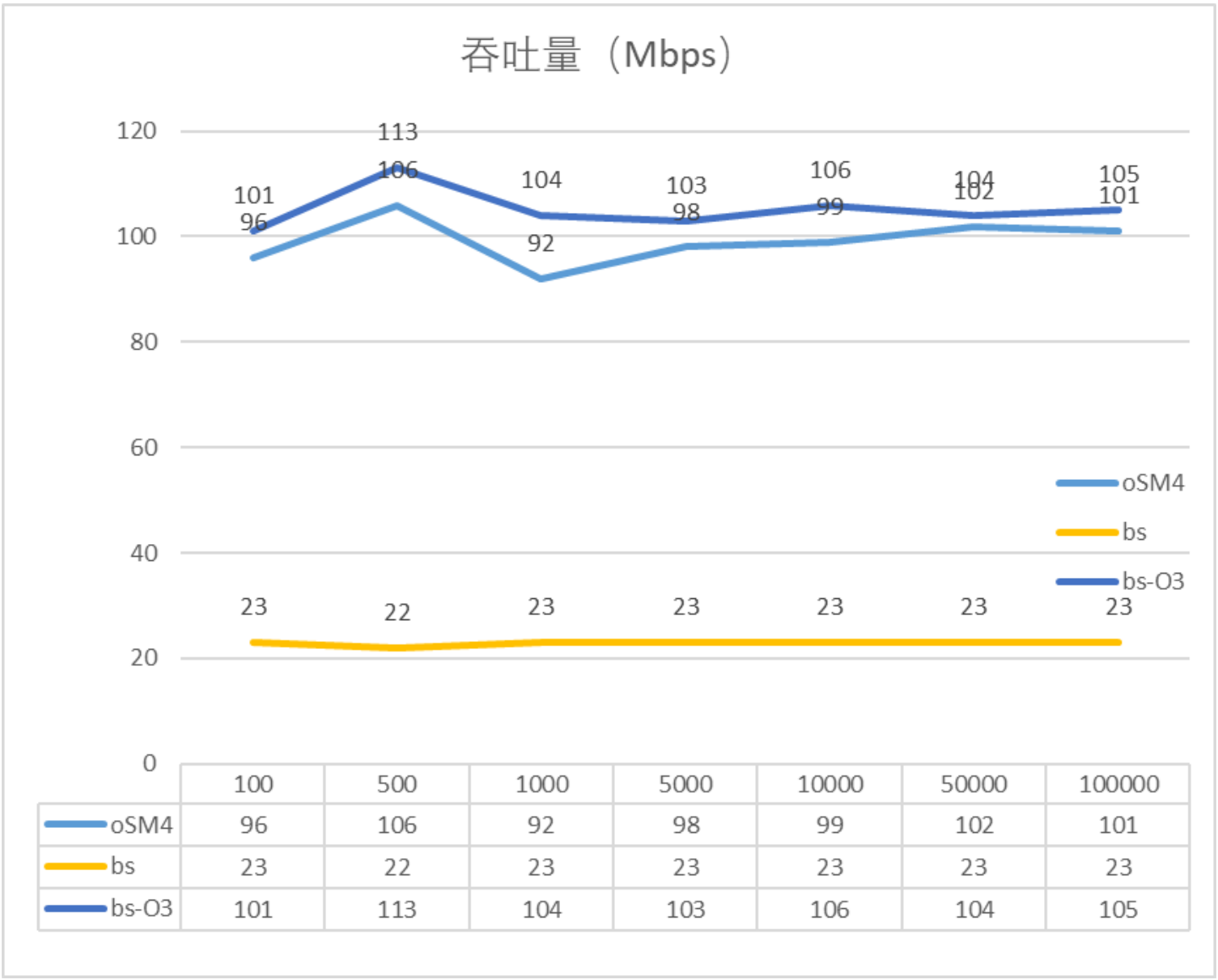


(a) The encryption time on the PC



(b) The encryption time on the edge node

对比计算，假如不考虑10^6 与2^20带来的误差，我们认定这里的M为10^6简化计算，则其左图吞吐量约为 $\frac{100MB \times 8B/b}{4s} = 200Mbps$，与我们当前版本的速率基本相近；优化版本测算是用的其自带的test文件，我们假定与论文一致，则发现这个数据大大不符。即使开O3优化，也仅与原初版本持平，多数据测试图如下：



	100	500	1000	5000	10000	50000	100000
oSM4	96	106	92	98	99	102	101
bs	23	22	23	23	23	23	23
bs-O3	101	113	104	103	106	104	105

实验分析

原因是什么呢？我们深入到源码中进行分析。

摘录一段函数进行分析：

```
void S_BOX_bs(unsigned long long cinh,unsigned long long cinl,unsigned long long
{
        unsigned char temp;
        th=0;
        th=th^((bit_4bit((cinh>>(3))&0x1111111111111111))&0xeeeeeeeeeeeeeeee);
        th=th^((bit_4bit((cinh>>(2))&0x1111111111111111))&0xffffffffffffffff);
        th=th^((bit_4bit((cinh>>(1))&0x1111111111111111))&0x7777777777777777);
        th=th^((bit_4bit((cinh>>(0))&0x1111111111111111))&0xbbbbbbbbbbbbbbbb);

        th=th^((bit_4bit((cinl>>(3))&0x1111111111111111))&0x5555555555555555);
        th=th^((bit_4bit((cinl>>(2))&0x1111111111111111))&0x2222222222222222);
        th=th^((bit_4bit((cinl>>(1))&0x1111111111111111))&0x9999999999999999);
        th=th^((bit_4bit((cinl>>(0))&0x1111111111111111))&0xcccccccccccccccc);

        tl=0;
        tl=tl^((bit_4bit((cinh>>(3))&0x1111111111111111))&0x5555555555555555);
        tl=tl^((bit_4bit((cinh>>(2))&0x1111111111111111))&0x2222222222222222);
        tl=tl^((bit_4bit((cinh>>(1))&0x1111111111111111))&0x9999999999999999);
        tl=tl^((bit_4bit((cinh>>(0))&0x1111111111111111))&0xcccccccccccccccc);

        tl=tl^((bit_4bit((cinl>>(3))&0x1111111111111111))&0xeeeeeeeeeeeeeeee);
        tl=tl^((bit_4bit((cinl>>(2))&0x1111111111111111))&0xffffffffffffffff);
        tl=tl^((bit_4bit((cinl>>(1))&0x1111111111111111))&0x7777777777777777);
        tl=tl^((bit_4bit((cinl>>(0))&0x1111111111111111))&0xbbbbbbbbbbbbbbbb);

        th=th^0xddddddddddddddd;
        tl=tl^0x3333333333333333;

        bit256_16X2_bs(th,tl,cinh,cinl);//这三个的后两个参数都是引用类型
        GF16_2_inver_bs(cinh,cinl,th,tl);
        bit16X2_256_bs(th,tl,cinh,cinl);
```

这段代码充分说明了，何为“理想很丰满，现实很骨感”，这个函数就可以管中窥豹太多的问题：

1. 使用引用变量，并直接存取：不管这里的引用究竟是在data或bss段的全局数据还是上一个调用函数的局部变量，都不是他原文中提到的充分利用寄存器；
2. 指令流依赖性强：可以发现前半都是th运算，后半都是tl运算，一半的时间都在同一个指令上浪费掉了；

3. 调用函数太随意：这几个函数都有引用类型，即使编译器试图优化也无法把部分变量送进寄存器中——寄存器存储不能跨函数，寄存器无地址可进行引用；
4. 大量的偷懒宏定义：这些宏定义并没有实际起到减少调用优化的作用，正相反，他甚至有一些可以合并的指令为了偷懒层层套娃了，这样无益于并行；

当然，由于对同一组数据进行多次加密和对不同数据进行单次加密在实际中还是有较大差异的，我们有理由相信在这里的循环计数方式与论文原文中提到的数据规模是有差异的，所以如果我们有处理海量数据的能力时，我们会再做另一版对照实验进行实验。

而在本次实验中，由于其文件体量过大，且依赖关系太过复杂，本人曾尝试优化了一版尽可能使用register的版本，但由于引用依赖问题失败了。想来这是别人的博士课题，也不是我小小一个本科生能在一周内处理的项目，故搁置，本次不再使用。

SIMD+流水线

原理分析

虽然bitslice失败了，但他仍启发了我们：能不能进行一些多路数据并行呢，这正是SIMD的工作。

因此我们在循环展开的优化上写了一版直接使用SIMD的多数据组并行，注意到SSE指令集有4*4转置函数，我们选择四轮展开。但是注意到，SIMD并没有针对256大小的查表指令，在这里我们别无他法，只能对每个8bits数据进行单独Sbox映射。

实验代码

```
void enc2(u32 m[],u32* rk)
{

    _MM_TRANSPOSE4_PS(*(__m128*)(m),*(__m128*)(m+4),*(__m128*)(m+8),*(__m128*)(m

    register __m128i
        tmpt,
        tmp1=_mm_load_si128(m),
        tmp2=_mm_load_si128(m+4),
        tmp3=_mm_load_si128(m+8),
        tmp4=_mm_load_si128(m+12);
    __m128i tmp0;
    u8 *t=(u8*)&tmp0;
    for(int i=0;i<32;i+=4)
    {
        register __m128i tmp,tmpd;

        tmp=_mm_set1_epi32(*rk);
        tmp=_mm_xor_si128(tmp,tmp4);
        tmpd=_mm_xor_si128(tmp2,tmp3);
        tmp0=_mm_xor_si128(tmpd,tmp);
```

```
22
23
24          t[ 0]=Sbox[t[ 0]];
25          t[ 1]=Sbox[t[ 1]];
26          t[ 2]=Sbox[t[ 2]];
27          t[ 3]=Sbox[t[ 3]];
28          t[ 4]=Sbox[t[ 4]];
29          t[ 5]=Sbox[t[ 5]];
30          t[ 6]=Sbox[t[ 6]];
31          t[ 7]=Sbox[t[ 7]];
32          t[ 8]=Sbox[t[ 8]];
33          t[ 9]=Sbox[t[ 9]];
34          t[10]=Sbox[t[10]];
35          t[11]=Sbox[t[11]];
36          t[12]=Sbox[t[12]];
37          t[13]=Sbox[t[13]];
38          t[14]=Sbox[t[14]];
39          t[15]=Sbox[t[15]];
40          rk++;
41          tmpt=tmp0;
42
43          tmp=_mm_slli_epi32(tmpt,2);
44          tmpd=_mm_srli_epi32(tmpt,30);
45          tmp1=_mm_xor_si128(tmp1,tmp);
46          tmp1=_mm_xor_si128(tmp1,tmpd);
47
48          //m3=tmp^rot(tmp,32,2)^rot(tmp,32,10)^rot(tmp,32,18)^rot(tmp,32,24)^m3;
49
50          tmp=_mm_slli_epi32(tmpt,10);
51          tmpd=_mm_srli_epi32(tmpt,22);
52          tmp1=_mm_xor_si128(tmp1,tmp);
53          tmp1=_mm_xor_si128(tmp1,tmpd);
54
55          tmp=_mm_slli_epi32(tmpt,18);
56          tmpd=_mm_srli_epi32(tmpt,14);
57          tmp1=_mm_xor_si128(tmp1,tmp);
58          tmp1=_mm_xor_si128(tmp1,tmpd);
59
60          tmp=_mm_slli_epi32(tmpt,24);
61          tmpd=_mm_srli_epi32(tmpt,8);
62          tmp1=_mm_xor_si128(tmp1,tmp);
63          tmp1=_mm_xor_si128(tmp1,tmpd);
64
65          tmp1=_mm_xor_si128(tmp1,tmpt);
66
67          tmp0=tmp1;
68
```

```
69
70          tmp=_mm_set1_epi32(*rk);
71          tmp=_mm_xor_si128(tmp,tmp4);
72          tmpd=_mm_xor_si128(tmp1,tmp3);
73          tmp0=_mm_xor_si128(tmpd,tmp);
74
75          t[ 0]=Sbox[t[ 0]];
76          t[ 1]=Sbox[t[ 1]];
77          t[ 2]=Sbox[t[ 2]];
78          t[ 3]=Sbox[t[ 3]];
79          t[ 4]=Sbox[t[ 4]];
80          t[ 5]=Sbox[t[ 5]];
81          t[ 6]=Sbox[t[ 6]];
82          t[ 7]=Sbox[t[ 7]];
83          t[ 8]=Sbox[t[ 8]];
84          t[ 9]=Sbox[t[ 9]];
85          t[10]=Sbox[t[10]];
86          t[11]=Sbox[t[11]];
87          t[12]=Sbox[t[12]];
88          t[13]=Sbox[t[13]];
89          t[14]=Sbox[t[14]];
90          t[15]=Sbox[t[15]];
91          rk++;
92          tmpt=tmp0;
93          tmp=_mm_slli_epi32(tmpt,2);
94          tmpd=_mm_srli_epi32(tmpt,30);
95          tmp2=_mm_xor_si128(tmp2,tmp);
96          tmp2=_mm_xor_si128(tmp2,tmpd);
97
98          tmp=_mm_slli_epi32(tmpt,10);
99          tmpd=_mm_srli_epi32(tmpt,22);
100         tmp2=_mm_xor_si128(tmp2,tmp);
101         tmp2=_mm_xor_si128(tmp2,tmpd);
102
103         tmp=_mm_slli_epi32(tmpt,18);
104         tmpd=_mm_srli_epi32(tmpt,14);
105         tmp2=_mm_xor_si128(tmp2,tmp);
106         tmp2=_mm_xor_si128(tmp2,tmpd);
107
108         tmp=_mm_slli_epi32(tmpt,24);
109         tmpd=_mm_srli_epi32(tmpt,8);
110         tmp2=_mm_xor_si128(tmp2,tmp);
111         tmp2=_mm_xor_si128(tmp2,tmpd);
112
113         tmp2=_mm_xor_si128(tmp2,tmpt);
114
115         tmp=_mm_set1_epi32(*rk);
```

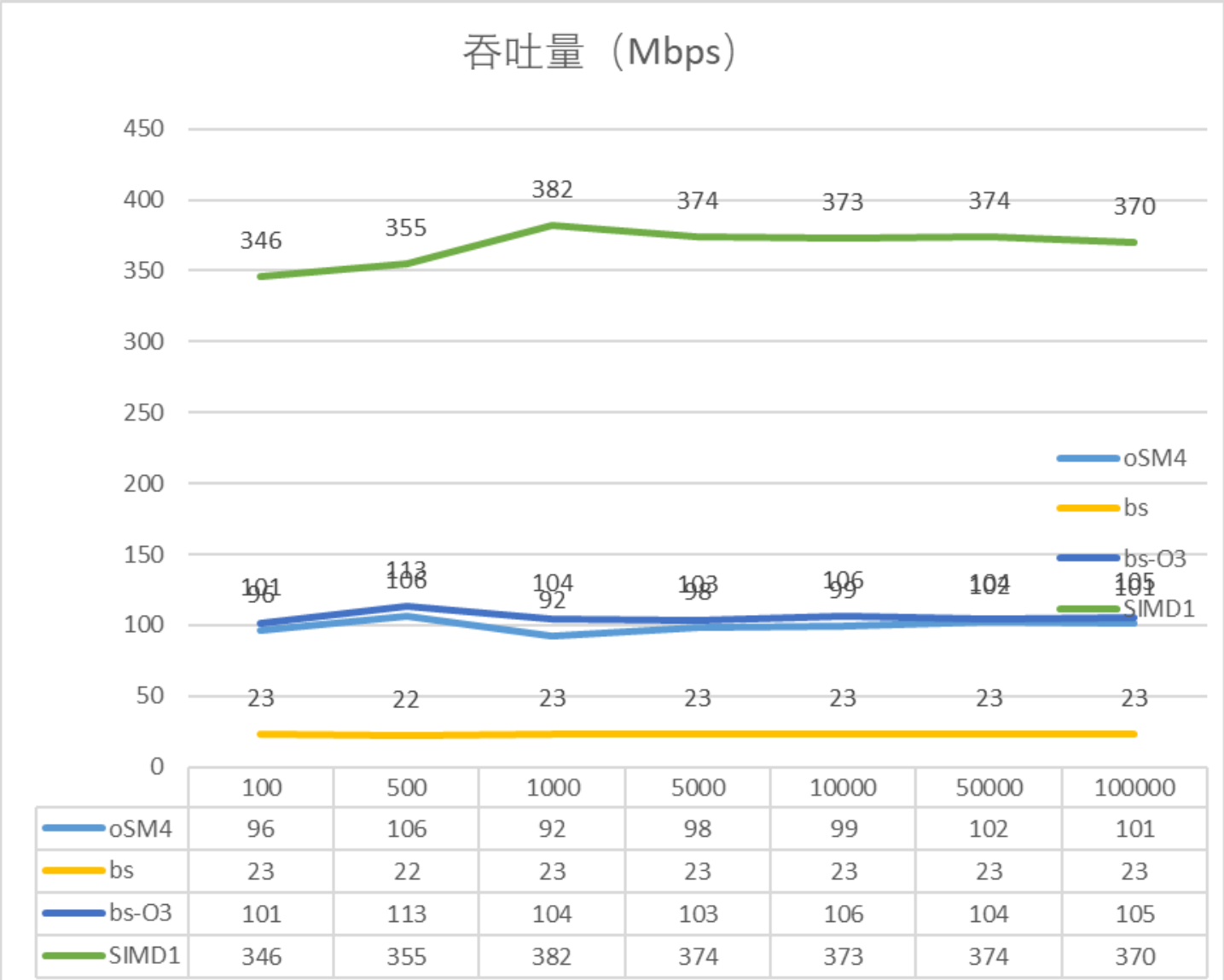
```
116         tmp=_mm_xor_si128(tmp,tmp4);
117         tmpd=_mm_xor_si128(tmp2,tmp1);
118         tmp0=_mm_xor_si128(tmpd,tmp);
119 
120         t[ 0]=Sbox[t[ 0]];
121         t[ 1]=Sbox[t[ 1]];
122         t[ 2]=Sbox[t[ 2]];
123         t[ 3]=Sbox[t[ 3]];
124         t[ 4]=Sbox[t[ 4]];
125         t[ 5]=Sbox[t[ 5]];
126         t[ 6]=Sbox[t[ 6]];
127         t[ 7]=Sbox[t[ 7]];
128         t[ 8]=Sbox[t[ 8]];
129         t[ 9]=Sbox[t[ 9]];
130         t[10]=Sbox[t[10]];
131         t[11]=Sbox[t[11]];
132         t[12]=Sbox[t[12]];
133         t[13]=Sbox[t[13]];
134         t[14]=Sbox[t[14]];
135         t[15]=Sbox[t[15]];
136         rk++;
137         tmpt=tmp0;
138         tmp=_mm_slli_epi32(tmpt,2);
139         tmpd=_mm_srli_epi32(tmpt,30);
140         tmp3=_mm_xor_si128(tmp3,tmp);
141         tmp3=_mm_xor_si128(tmp3,tmpd);
142 
143         tmp=_mm_slli_epi32(tmpt,10);
144         tmpd=_mm_srli_epi32(tmpt,22);
145         tmp3=_mm_xor_si128(tmp3,tmp);
146         tmp3=_mm_xor_si128(tmp3,tmpd);
147 
148         tmp=_mm_slli_epi32(tmpt,18);
149         tmpd=_mm_srli_epi32(tmpt,14);
150         tmp3=_mm_xor_si128(tmp3,tmp);
151         tmp3=_mm_xor_si128(tmp3,tmpd);
152 
153         tmp=_mm_slli_epi32(tmpt,24);
154         tmpd=_mm_srli_epi32(tmpt,8);
155         tmp3=_mm_xor_si128(tmp3,tmp);
156         tmp3=_mm_xor_si128(tmp3,tmpd);
157 
158         tmp3=_mm_xor_si128(tmp3,tmpt);
159 
160         tmp=_mm_set1_epi32(*rk);
161         tmp=_mm_xor_si128(tmp,tmp1);
162         tmpd=_mm_xor_si128(tmp2,tmp3);
```

```
163         tmp0=_mm_xor_si128(tmpd,tmp);
164 
165         t[ 0]=Sbox[t[ 0]];
166         t[ 1]=Sbox[t[ 1]];
167         t[ 2]=Sbox[t[ 2]];
168         t[ 3]=Sbox[t[ 3]];
169         t[ 4]=Sbox[t[ 4]];
170         t[ 5]=Sbox[t[ 5]];
171         t[ 6]=Sbox[t[ 6]];
172         t[ 7]=Sbox[t[ 7]];
173         t[ 8]=Sbox[t[ 8]];
174         t[ 9]=Sbox[t[ 9]];
175         t[10]=Sbox[t[10]];
176         t[11]=Sbox[t[11]];
177         t[12]=Sbox[t[12]];
178         t[13]=Sbox[t[13]];
179         t[14]=Sbox[t[14]];
180         t[15]=Sbox[t[15]];
181         rk++;
182         tmpt=tmp0;
183         tmp=_mm_slli_epi32(tmpt,2);
184         tmpd=_mm_srli_epi32(tmpt,30);
185         tmp4=_mm_xor_si128(tmp4,tmp);
186         tmp4=_mm_xor_si128(tmp4,tmpd);
187 
188         tmp=_mm_slli_epi32(tmpt,10);
189         tmpd=_mm_srli_epi32(tmpt,22);
190         tmp4=_mm_xor_si128(tmp4,tmp);
191         tmp4=_mm_xor_si128(tmp4,tmpd);
192 
193         tmp=_mm_slli_epi32(tmpt,18);
194         tmpd=_mm_srli_epi32(tmpt,14);
195         tmp4=_mm_xor_si128(tmp4,tmp);
196         tmp4=_mm_xor_si128(tmp4,tmpd);
197 
198         tmp=_mm_slli_epi32(tmpt,24);
199         tmpd=_mm_srli_epi32(tmpt,8);
200         tmp4=_mm_xor_si128(tmp4,tmp);
201         tmp4=_mm_xor_si128(tmp4,tmpd);
202 
203         tmp4=_mm_xor_si128(tmp4,tmpt);
204 
205     }
206     _mm_storeu_si128(m,tmp1);
207     _mm_storeu_si128(m+4,tmp2);
208     _mm_storeu_si128(m+8,tmp3);
209     _mm_storeu_si128(m+12,tmp4);
```

```
210
211     _MM_TRANSPOSE4_PS(*(__m128*)(m+12),*(__m128*)(m+8),*(__m128*)(m+4),*(__m128*
212 }
```

实验效果

实验效果并没有变得更好：比原初版本提升了四倍左右，但慢于基础优化与循环展开。

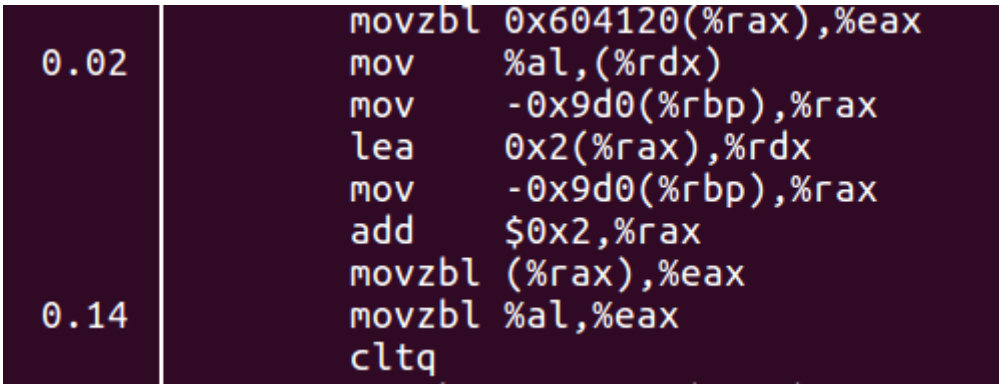


	100	500	1000	5000	10000	50000	100000
oSM4	96	106	92	98	99	102	101
bs	23	22	23	23	23	23	23
bs-O3	101	113	104	103	106	104	105
SIMD1	346	355	382	374	373	374	370

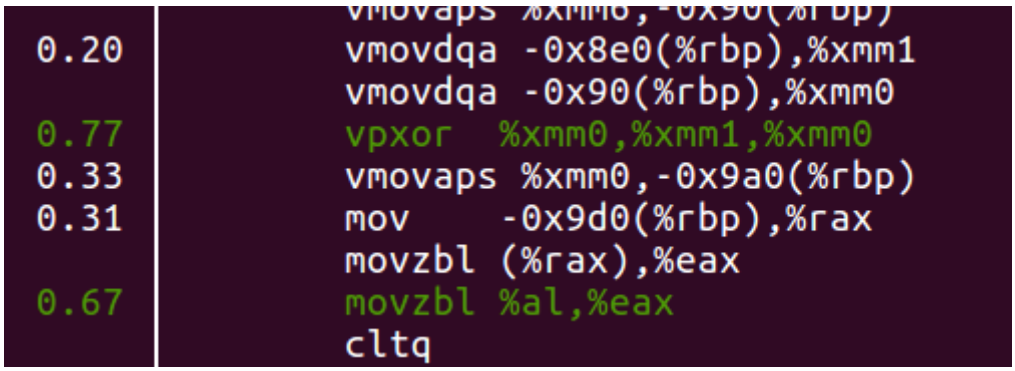
结果分析

为什么展开数据进行SIMD之后反而更慢了呢，我们使用perf对时钟周期进行跟踪，发现了如下情况：

1. 程序展开后有大量的如下的块，分析功能后发现为查Sbox的基本单元，这种基本单元每一个都用去了大概0.2%的时间，每轮有16个基本块，展开了四轮，所以大概用去13%左右的时间；

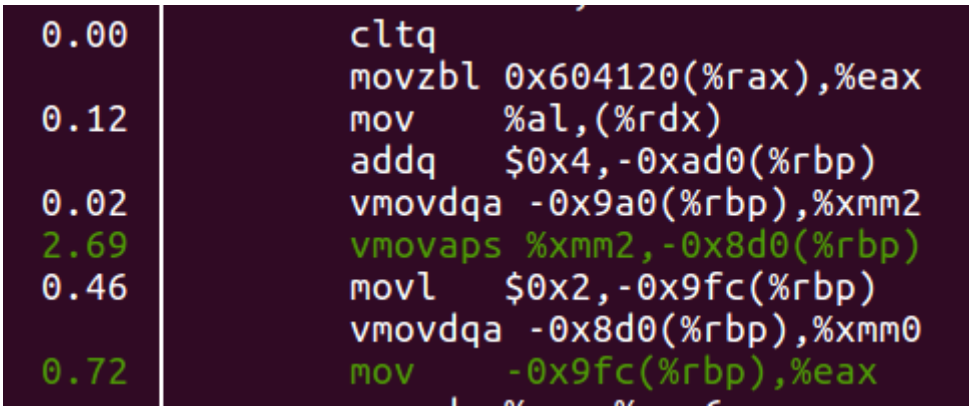


2. 在查表开始的阶段，有这么一块



这里的xor到cltq之间都是为了能将数据按位取出来而从寄存器放入栈中的过程，可以发现大概占用2%，四轮即8%；

3. 在查表结束的阶段，有这么一块



将查表数据读回寄存器，又花费了3%，四轮即12%；

4. 除此之外，我的电脑支持ror指令，而SIMD没有类似的功能，所以也有影响。
5. 还有register作为指令只是建议函数使用寄存器，但我们仍可以发现有一些变量还是声明为了栈上局部变量，对他们的访存影响仍较大。

```
                vmovdqa -0x6f0(%rbp),%xmm1
 0.69           vmovdqa -0x1b0(%rbp),%xmm0
                vpxor   %xmm0,%xmm1,%xmm0
 0.36           vmovdqa %xmm0,%xmm5
                vmovaps %xmm5,-0x6e0(%rbp)
 0.32           vmovdqa %xmm15,%xmm4
                vmovaps %xmm4,-0x1c0(%rbp)
                vmovdqa -0x6e0(%rbp),%xmm1
 0.84           vmovdqa -0x1c0(%rbp),%xmm0
                vpxor   %xmm0,%xmm1,%xmm0
 0.29           vmovdqa %xmm0,%xmm1
```

所以总体上，查表就占用了33%左右的性能，再加上不好估量的寄存器问题，我们可以说这里有至少50%的时间与我们预期的行为不符从而造成了浪费，我们受其制约，而这是在现阶段指令集下无法避免的。如果能对其优化，则至少能提升到与循环展开同等的水平。

综合优化

基本思路

前文我们分别讨论了多线程，循环展开，SIMD和查表四种优化，最后的制约点落在了查表上。那么综合起来，有没有什么方案能解决这个瓶颈呢?

有。注意到在AVX2指令集中，i32gather开始支持查大表。但由于AVX2主要支持256bits,这时就要求我们至少需要八路并行。

前文查大表的方式可以优化掉左右移两次计算的问题。

另外，查表时我们只需要查部分位，这样我们可以通过and mask的传统方式进行。

具体实现

为形象直观地分析，我们绘制表格分析内存：

直接load内存时，我们会load出如下左情况。

		0						7
T1		A1	A2	A3	A4	B1	B2	B3	B4
T2		C1	C2	C3	C4	D1	D2	D3	D4
T3		E1	E2	E3	E4	F1	F2	F3	F4
T4		G1	G2	G3	G4	H1	H2	H3	H4

A1	B1	C1	D1	E1	F1	G1	H1
A2	B2	C2	D2	E2	F2	G2	H2
A3	B3	C3	D3	E3	F3	G3	H3
A4	B4	C4	D4	E4	F4	G4	H4

但我们需要上右的情况。注意到其序号二进制表示中最高位并不影响，而unpack函数正好支持按照64和32进行划分打包，于是进行打包：

```
#define MM256_PACK1_EPI32(a, b, c, d)                  \
    _mm256_unpacklo_epi64(_mm256_unpacklo_epi32(a, b), \
                          _mm256_unpacklo_epi32(c, d))
#define MM256_PACK2_EPI32(a, b, c, d)                  \
    _mm256_unpackhi_epi64(_mm256_unpacklo_epi32(a, b), \
                          _mm256_unpacklo_epi32(c, d))
#define MM256_PACK3_EPI32(a, b, c, d)                  \
    _mm256_unpacklo_epi64(_mm256_unpackhi_epi32(a, b), \
                          _mm256_unpackhi_epi32(c, d))
#define MM256_PACK4_EPI32(a, b, c, d)                  \
    _mm256_unpackhi_epi64(_mm256_unpackhi_epi32(a, b), \
                          _mm256_unpackhi_epi32(c, d))

```

便可以调整为右图模式。

实现代码

```
void _SM4_(uint32_t* in, uint32_t* out, SM4_Key sm4_key, int enc) {
    register __m256i X1,X2,X3,X4, T1,T2,T3,T4, Mask;
    Mask = _mm256_set1_epi32(0xFF);

    T1 = _mm256_loadu_si256((const __m256i*)in + 0);
    T2 = _mm256_loadu_si256((const __m256i*)in + 1);
    T3 = _mm256_loadu_si256((const __m256i*)in + 2);
    T4 = _mm256_loadu_si256((const __m256i*)in + 3);

    X1 = MM256_PACK0_EPI32( T1, T2, T3,T4);
    X2 = MM256_PACK1_EPI32( T1, T2, T3,T4);
    X3 = MM256_PACK2_EPI32( T1, T2, T3,T4);
    X4 = MM256_PACK3_EPI32( T1, T2, T3,T4);
    for (int i = 0; i < 32; i+=4) {
        register __m256i k =
            _mm256_set1_epi32((enc == 0) ? sm4_key[i] : sm4_key[31 - i]);
        T1 = _mm256_xor_si256(_mm256_xor_si256(X2, X3),
                              _mm256_xor_si256(X4, k));
        T2 = _mm256_xor_si256(
            X1, _mm256_i32gather_epi32((const int*)BOX0,
                                       _mm256_and_si256(T1, Mask), 4));
        T1 = _mm256_srli_epi32(T1, 8);
        T2 = _mm256_xor_si256(
            T2, _mm256_i32gather_epi32(
                        (const int*)BOX1, _mm256_and_si256(T1, Mask), 4));
        T1 = _mm256_srli_epi32(T1, 8);
        T2 = _mm256_xor_si256(
```

```
            T2, _mm256_i32gather_epi32(
                        (const int*)BOX2, _mm256_and_si256(T1, Mask), 4));
        T1 = _mm256_srli_epi32(T1, 8);
        T2 = _mm256_xor_si256(
            T2, _mm256_i32gather_epi32(
                        (const int*)BOX3, _mm256_and_si256(T1, Mask), 4));
        X1=T2;

        k=    _mm256_set1_epi32((enc == 0) ? sm4_key[i+1] : sm4_key[30 - i]);
        T1 = _mm256_xor_si256(_mm256_xor_si256(X1, X3),
                                _mm256_xor_si256(X4, k));
        T2 = _mm256_xor_si256(
            X2, _mm256_i32gather_epi32((const int*)BOX0,
                                        _mm256_and_si256(T1, Mask), 4));
        T1 = _mm256_srli_epi32(T1, 8);
        T2 = _mm256_xor_si256(
            T2, _mm256_i32gather_epi32(
                        (const int*)BOX1, _mm256_and_si256(T1, Mask), 4));
        T1 = _mm256_srli_epi32(T1, 8);
        T2 = _mm256_xor_si256(
            T2, _mm256_i32gather_epi32(
                        (const int*)BOX2, _mm256_and_si256(T1, Mask), 4));
        T1 = _mm256_srli_epi32(T1, 8);
        T2 = _mm256_xor_si256(
            T2, _mm256_i32gather_epi32(
                        (const int*)BOX3, _mm256_and_si256(T1, Mask), 4));
        X2=T2;

        k=    _mm256_set1_epi32((enc == 0) ? sm4_key[i+2] : sm4_key[29 - i]);
        T1 = _mm256_xor_si256(_mm256_xor_si256(X1, X2),
                                _mm256_xor_si256(X4, k));
        T2 = _mm256_xor_si256(
            X3, _mm256_i32gather_epi32((const int*)BOX0,
                                        _mm256_and_si256(T1, Mask), 4));
        T1 = _mm256_srli_epi32(T1, 8);
        T2 = _mm256_xor_si256(
            T2, _mm256_i32gather_epi32(
                        (const int*)BOX1, _mm256_and_si256(T1, Mask), 4));
        T1 = _mm256_srli_epi32(T1, 8);
        T2 = _mm256_xor_si256(
            T2, _mm256_i32gather_epi32(
                        (const int*)BOX2, _mm256_and_si256(T1, Mask), 4));
        T1 = _mm256_srli_epi32(T1, 8);
        T2 = _mm256_xor_si256(
            T2, _mm256_i32gather_epi32(
                        (const int*)BOX3, _mm256_and_si256(T1, Mask), 4));
        X3=T2;
```

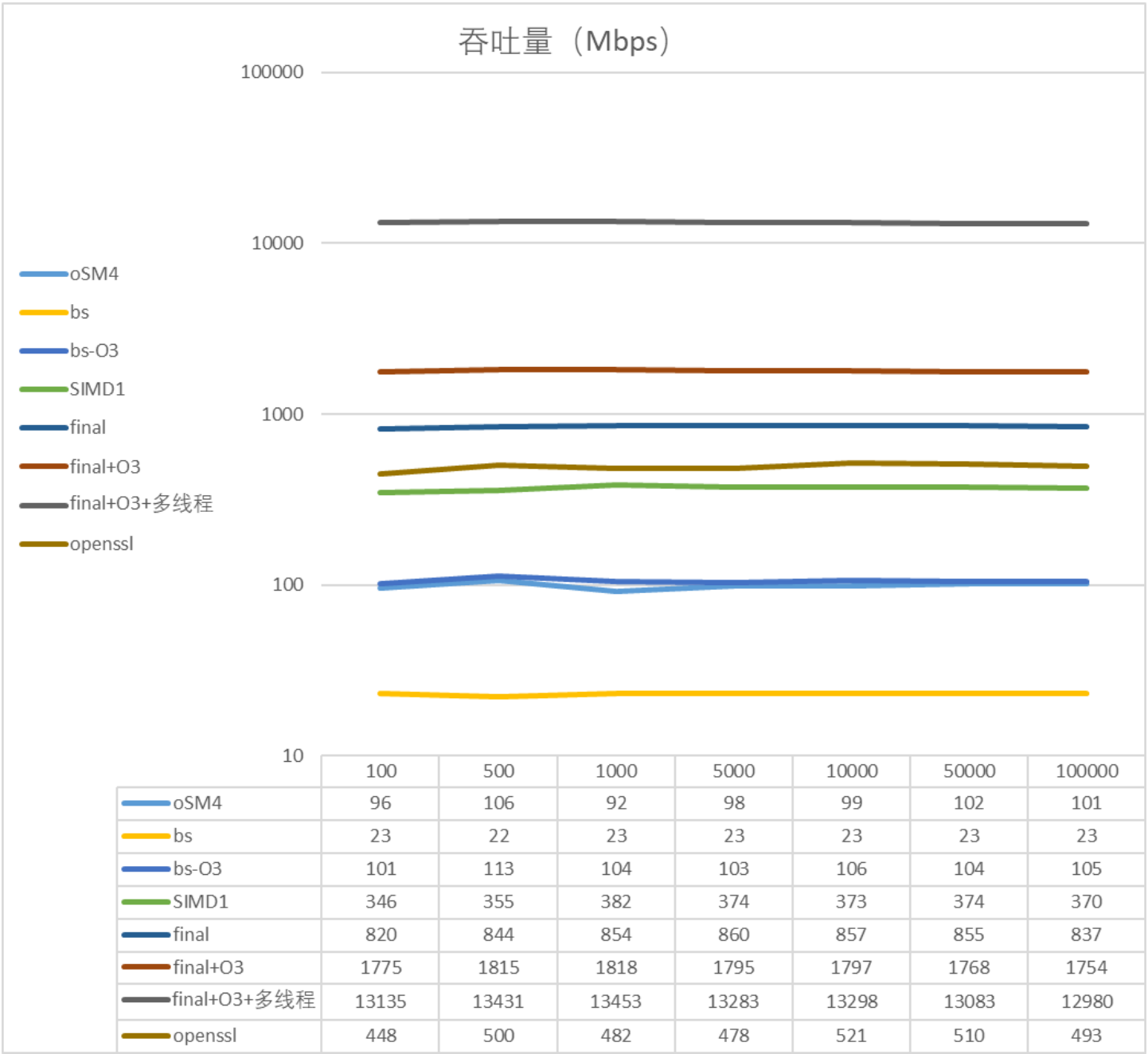
```
75
76
77          k=    _mm256_set1_epi32((enc == 0) ? sm4_key[i+3] : sm4_key[28 - i]);
78          T1 = _mm256_xor_si256(_mm256_xor_si256(X1, X3),
79                                        _mm256_xor_si256(X2, k));
80          T2 = _mm256_xor_si256(
81              X4, _mm256_i32gather_epi32((const int*)BOX0,
82                                         _mm256_and_si256(T1, Mask), 4));
83          T1 = _mm256_srli_epi32(T1, 8);
84          T2 = _mm256_xor_si256(
85              T2, _mm256_i32gather_epi32(
86                           (const int*)BOX1, _mm256_and_si256(T1, Mask), 4));
87          T1 = _mm256_srli_epi32(T1, 8);
88          T2 = _mm256_xor_si256(
89              T2, _mm256_i32gather_epi32(
90                           (const int*)BOX2, _mm256_and_si256(T1, Mask), 4));
91          T1 = _mm256_srli_epi32(T1, 8);
92          T2 = _mm256_xor_si256(
93              T2, _mm256_i32gather_epi32(
94                           (const int*)BOX3, _mm256_and_si256(T1, Mask), 4));
95          X4=T2;
96      }
97      _mm256_storeu_si256((__m256i*)out + 0,
98                          MM256_PACK0_EPI32(X4, X3, X2, X1));
99      _mm256_storeu_si256((__m256i*)out + 1,
100                         MM256_PACK1_EPI32(X4, X3, X2, X1));
101     _mm256_storeu_si256((__m256i*)out + 2,
102                         MM256_PACK2_EPI32(X4, X3, X2, X1));
103     _mm256_storeu_si256((__m256i*)out + 3,
104                         MM256_PACK3_EPI32(X4, X3, X2, X1));
105 }
```

该代码规避了前文提到的大部分问题。

实验效果

速度也达到并超过了上文SIMD的预期。

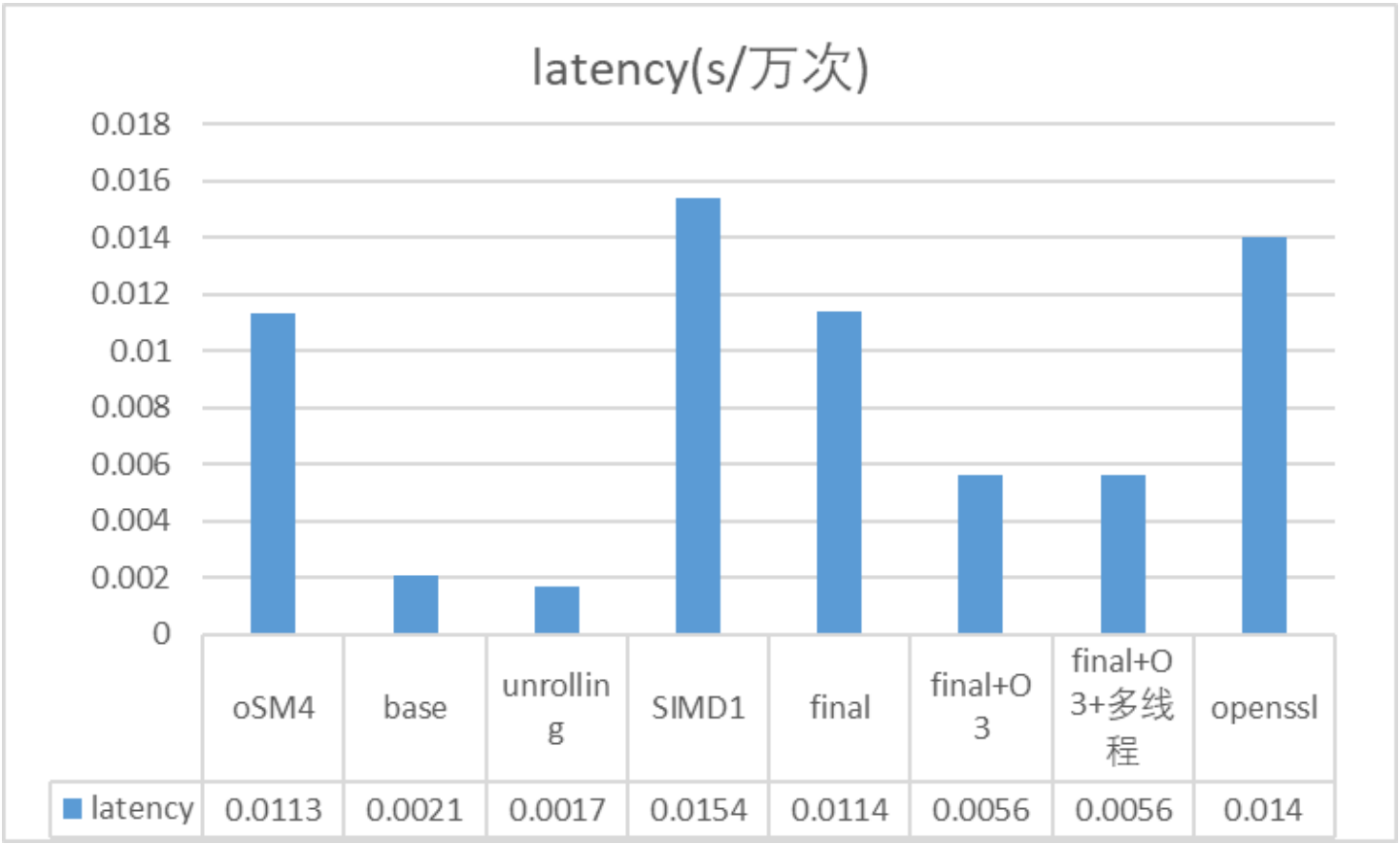
开启O3以及加入8核多线程后得到最终结果为



	100	500	1000	5000	10000	50000	100000
oSM4	96	106	92	98	99	102	101
bs	23	22	23	23	23	23	23
bs-O3	101	113	104	103	106	104	105
SIMD1	346	355	382	374	373	374	370
final	820	844	854	860	857	855	837
final+O3	1775	1815	1818	1795	1797	1768	1754
final+O3+多线程	13135	13431	13453	13283	13298	13083	12980
openssl	448	500	482	478	521	510	493

数据分析

由于延时是通过每次执行的时间测量，但实际上多次之间是相对独立的，我们这里用每万次的时间进行分析。



由于那两个bitslice的值实在太大，在做表格时我们放弃了他们。可以发现，虽然SIMD指令总体上优化了时间，增加了吞吐量，但由于处理的指令较长，实际的latency也更大；而普通处理过的特别是进行了循环展开的SM4很明显更快一些，延时也更小。

在ub18及以上的版本中，openssl自带sm4，按照同等方式调用接口得到记录如上图，可以发现，openssl的时延与SIMD相仿；总体的吞吐量也与SIMD版本相仿。

结论

在吞吐量上，我们的最终版本大约是原初版本的135倍，是最慢版本的4250倍，是openssl的26倍；

在延时上，我们的最终版本大约是原初版本的5.4倍，是最慢版本的2.85%，是openssl的81%；最佳版本是原初版本的81%，是最慢版本的0.4%，是openssl的12%。

可以看出，查表优化是通用的优化；在数据量较小或并行度较差的工作模式时可只使用循环展开，而不需要使用其他方案；而在数据量较大或并行度较高的工作模式时加上SIMD技术和多线程可增大数据吞吐量；对于流水线上的bitslice等优化，暂时还没看到较好的SM4软件实现，但参考2006年对AES的bitslice优化，其bitslice版本相比于当时最好的软件优化版本时钟周期数减少到1/3左右，说明合理的bitslice优化还是在抗侧信道之外有优化空间的，具体希望可以在以后的日子里做进一步探讨。

实验参考资料

FPGA静态时序分析系列博文(目录篇)-Felix-电子技术应用-AET-中国科技核心期刊-最丰富的电子设计资源平台

系统级性能分析工具perf的介绍与使用[转] - sunsky303 - 博客园

SM4的快速软件实现技术

SM4 Bit Slice 实现